C10

```c
 2   /********************************************************************
 3   **
 4   ** File Name: RSLinitfin.c
 5   **
 6   ** Copyright (c) 1998,1999 by EMC Corporation.
 7   **
 8   ** Purpose:
 9   **          This module contains the Restore Service Library functions to
10   **          initialize and terminate the restore operation.
11   **
12   **
13   ** Table of Contents:
14   **
15   **          RSTSL_Initialize
16   **          RSTSL_Finish
17   **
18   **          Internal Functions:
19   **
20   **
21   ** Compile-Time Options:
22   **          This section must list any compile time definitions
23   **          which will affect this header.
24   **
     ********************************************************************/

27   /* The following provides an RCS id in the binary that can be located
28   ** with the what(1) utility.  The intent is to keep this short.
29   */

31   #ifndef lint
32   static char RCS_id [] = "$RCSfile$ "
33                           "$Revision$ "
34                           "$Date$" ;
35   #endif

38   /*
39    * Feature test switches.
40    * Standard defines required to turn on OS features go here.
41    *
42    * The following is required for code that uses POSIX API's.
43    * Remove for non-POSIX, non-portable code.
44    */

46   #define _POSIX_SOURCE 1

49   /*
50    * System headers.
51    */
52   #include <sys/param.h>
53   #include <dirent.h>
54   #include <dlfcn.h>

57   /*
58    * Epoch headers.
59    */
60   #include <eb/eb_port.h>
61   #include <eb/rb_log.h>
```

```c
64   /*
65    * Local headers
66    */
67   #include <RSLinterns.h>
68   #include <RSLstartup.h>

71   /*
72    * #defines, structures, typedefs local to this source file
73    */

75   static eerrno_ty init_plugins( restore_context *rcp );
76   static int validate_plugin( struct pluginData *piDataPtr );

79   /*
80    * External declarations
81    */

83   /*
84    * This is the global "restore context" that will be used
85    * throughout the rest of the restore operations.
86    */
87   struct restore_context *rcp = NULL;

89   /*
90    * Definitions of the names of the plugin functions in the piFuncArray
91    * of the pluginData structure.        These must be in the same order and position
92    * as the piFuncIndex values defined in RSLplugin.h.
93    */
94   char *piFuncNames[PIFuncIndexLast+1] = {
95        "RSTPI_Identify",
96        "RSTPI_Initialize",
97        "RSTPI_GetTopLevelObjects",
98        "RSTPI_SetTopLevelObject",
99        "RSTPI_GetNextLevelObjects",
100       "RSTPI_ClearRestoreContext",
101       "RSTPI_Submit",
102       "RSTPI_GetTopLevelTemplates",
103       "RSTPI_DoesAlternateExist",
104       "RSTPI_MarkObject",
105       "RSTPI_UnmarkObject",
106       "RSTPI_IsObjectMarked",
107       "RSTPI_IsObjectMarkable",
108       "RSTPI_GetAllBackupTimes",
109       "RSTPI_GetCurrentBackupTime",
110       "RSTPI_SetBackupForTime",
111       "RSTPI_SetNextBackup",
112       "RSTPI_SetPrevBackup",
113       "RSTPI_SetFirstBackup",
114       "RSTPI_SetMostRecentBackup",
115       "RSTPI_IsTherePrevBackup",
116       "RSTPI_IsThereNextBackup",
117       "RSTPI_IsTherePrevBackupForTime",
118       "RSTPI_IsThereNextBackupForTime",
119       "RSTPI_Finish",
120       "RSTPI_StartRestore",
121       "RSTPI_FindRestorableObjects",
122       "RSTPI_GetFindResults",
123       "RSTPI_GetNecessaryMedia"
124  };
```

```
127
128
129
130  1  /* *********************************************************
131  1   *
132  1   * RSTSL_Initialize:
133  1   *
134  1   *   This function takes care of all the initialization for a restore
135  1   *   session.  This must be called prior to any of the other functions
136  1   *   in the Restore API.
137  1   *
138  1   * Parameters:
140  1   *
141  1   *   userName (I) - The name of the user.
142  1   *
143  1   * *********************************************************/
        eerrno_ty
        RSTSL_Initialize( const char *userName )
        {
            eerrno_ty status = E_SUCCESS;
145  1      /*
146  1       * If we have not yet allocated space for a restore_context
147  1       * structure, do so now.  If we have already done so,  just clear it
148  1       * now.
149  1       */
151  1      if (NULL == rcp)
152  2      {
153  2          rcp = (struct restore_context *)malloc(sizeof(
                    struct restore_context));
154  2          if (NULL == rcp)
155  3          {
156  3              rec_api_log_csm(SUB_CSM_NOMEM, NULL);
157  3              return(EP_RB_RECOVER_NOMEM);
158  2          }
159  1      }
160  1      memset(rcp, 0, sizeof(struct restore_context));
162  1      rcp->rc_human_uidname = esl_strdup( userName );
165  2      if (!rcp->rc_human_uidname) {
166  2          rec_api_log_csm(SUB_CSM_NOMEM, NULL);
167  2          return(EP_RB_RECOVER_NOMEM);
168  1      }
170  1      /*
171  1       * Set the appropriate field in the recovery context to indicate
172  1       * that this recover session is based on the Recover API.
173  1       * This flag is in place for historical reasons but is used by
174  1       * other parts of the Recover API library.
175  1       */
177  1      rcp->gui_mode = 1;
179  1      /*
180  1       * Initialize the logging mechanism.
181  1       */
183  1      if (status = rbrlog_begin(rcp, progname))
184  2      {
185  2          return(status);
186  1      }
188  1      /*
189  1       * Initialize the few "recover context" variables that we can at
190  1       * this early stage.
```

```
191  1       */
193  1      setup_proc(rcp);
195  1      /*
196  1       * The following call will:
197  1       *   -Initialize the saveset datatbase.
198  1       *   -Infer any information we can at this point.
199  1       */
201  1      if (status = startup(rcp))
202  2      {
203  2          return(status);
204  1      }
206  1      /* Do plugins setup: Find and initialize all valid restore plugin
                libs: */
208  1      status = init_plugins( rcp ) ;
210  1      return( status ) ;
                   /* End of RSTSL_Initialize() */
211  1  }
```

```c
214      /**************************************************************
215      *
216      *  RSTSL_Finish
217      *
218      *  Function Description:
219      *
220      *  This function terminates a restoral session,
         *                            but not while a restore is in
221      *  progress.
222      *     It will be rejected if a restore is currently being executed.
223      *  This routine will clean up any local memory used in the session.
224      *
225      *  Parameters:
226      *
227      *  none
         *
         *
229      */
230      eerrno_ty
231      RSTSL_Finish( void )
232   1  {
235   1      int mc_n;

237   1      eerrno_ty err = E_SUCCESS;

238   2      RemoveSubmitFiles();

239   2      if (NULL == rcp)
240   1      {
241   1          return( E_SUCCESS ) ;
242   1      }

243   1      /*
244   1       * Call rbr_cleanup() which kills the aux proc
245   1       * item, then calls rbrlog_end(
             *                              s), unlocks the work
             *                       ) to enter the last logs and to close
246   1       * the log file.
             */

248   1      rbr_cleanup(rcp);

250   1      /*
251   1       * Deallocate the memory of restore_context and the related
252   1       * structures.
253   1       */

255   1      if (NULL != rcp->rc_mcp)          /* Free the multicat structures */
256   2      {
257   2          mcat_destroy(rcp->rc_mcp) ;
258   1      }

260   1      /*
261   1       * Free the mark bit map space
262   1       */

264   2      for (mc_n = 0; mc_n < rcp->rc_marks_plane_alloc; mc_n++)
265   2      {
266   2          if (NULL != rcp->rc_marks[mc_n])
267   3          {
268   3              free(rcp->rc_marks[mc_n]) ;

270   2              rcp->rc_marks[mc_n] = NULL;
271   1          }
          }

273   1      if (NULL != rcp->rc_nmarks_by_plane)
```

```c
274   2      {
275   2          free(rcp->rc_nmarks_by_plane) ;
276   1      }

278   1      /*
279   1       * Free the configuration structures
280   1       */

282   1  #if 0
283   1      if (NULL != rcp->rc_cfgname)
284   2      {
285   2          free(rcp->rc_cfgname) ;
286   2      }
287   2  #endif

289   1      if (NULL != rcp->rc_config)
290   2      {
291   2          rbc_freeconfig(rcp->rc_config) ;
292   1      }

294   1      /*
295   1       * Free the DS_NONE structures array
296   1       * Note that even though rc_dsnones is the head of linked list
297   1       * of dsnone_info structures, the list is allocated via malloc
298   1       * as an array initially (ref. alloc_plane_arrays()), therefore
299   1       * we can do a free here.
300   1       */

302   2      if (NULL != rcp->rc_dsnones)
303   2      {
304   2          free(rcp->rc_dsnones) ;
305   1      }

307   1      /*
308   1       * Free the volume list structures.
309   1       */

311   1      if (NULL != rcp->ebvllist)
312   2      {
313   2          (void)ebvl_volidlist_destructor(
314   1                  rcp->ebvllist, EBVL_DESTROY_ALL);
          }

316   1      /*
317   1       * Free the plugin related data
318   1       */

320   1      rcp->rc_backup_app = 0;
321   2      while (rcp->currentPIptr = rcp->pilist)
322   2      {
323   2          rcp->rc_backup_app++;
324   2          rcp->appData = rcp->currentPIptr->appData;
325   2          /* allow plugin to clean up and close .so: */
326   2          if ( E_SUCCESS != (err =
327   2              rcp-> currentPIptr-> piFuncArray[ PIFuncIndexFinish ] (
328   3                  (struct pluginIDdata *)(
329   3                  rcp-> currentPIptr-> idData)) ) )
330   3          {
331   3              /* log error, continue */
332   3              rbe_user_error( err,
333   2                  "RSTPI_Finish failed for restore plug-in
                         library %s\n",
                         rcp-> currentPIptr-> name ) ;
          }
334   2      dlclose( rcp-> currentPIptr-> libHdl ) ;
335   2      rcp->pilist = rcp->pilist->next;
```

```
336  2          free (rcp->currentPIptr) ;
337  1      }
339  1
340  1      /*
341  1       * Free the various simple string buffers
343  1       */
344  2      if (NULL != rcp->rc_top_level_object_name)
345  2      {
346  1          free(rcp->rc_top_level_object_name);
348  1      }
349  2      if (NULL != rcp->rc_template_name)
350  2      {
351  1          free(rcp->rc_template_name);
353  1      }
354  2      if (NULL != rcp->rc_workitem_name)
355  2      {
356  1          free(rcp->rc_workitem_name);
358  1      }
359  2      if (NULL != rcp->rc_effective_uidname)
360  2      {
361  1          /* don't free, its internal: free(rcp->rc_effective_uidname); */
363  1      }
364  2      if (NULL != rcp->rc_human_uidname)
365  2      {
366  1          free(rcp->rc_human_uidname);
368  1      }
369  2      if (NULL != rcp->rc_client_rbuname)
370  2      {
371  1          free(rcp->rc_client_rbuname);
373  1      }
374  2      if (NULL != rcp->rc_client_hostname)
375  2      {
376  1          free(rcp->rc_client_hostname);
378  1      }
379  2      if (NULL != rcp->rc_client_scriptname)
380  2      {
381  1          /* don't free, its internal: free(rcp->rc_client_scriptname); */
383  1      }
384  2      if (NULL != rcp->rc_client_dirtop)
385  2      {
386  1          free(rcp->rc_client_dirtop);
388  1      }
389  2      if (NULL != rcp->rc_cmd_context)
390  2      {
391  1          /* don't free -- its internal/temp data: free(
                     rcp->rc_cmd_context); */
393  1      }
394  2      if (NULL != rcp->rc_source_client_hostname)
395  2      {
396  1          free(rcp->rc_source_client_hostname);
398  1      }
           if (NULL != rcp->rc_cpiogen_executable)
```

```
399  2      {
400  2          /* don't free, its internal: free(rcp->rc_cpiogen_executable); */
401  1      }
403  1      if (NULL != rcp->rc_plugin_wi_types)
404  2      {
405  2          free(rcp->rc_plugin_wi_types);
406  1      }
408  1      if (NULL != rcp->rc_pwd)
409  2      {
410  2          free(rcp->rc_pwd);
411  1      }
413  1      /*
414  1       * Finally, deallocate the restore_context itself
415  1       */
417  1      free(rcp);
418  1      rcp = NULL;
422  1      return( err );    /* RSTSL_Finish */
423  1  }
```

```
427    /***********************************************************
428    *
429    *   init_plugins
430    *
431    *   Function Description:
432    *
433    *   This function locates, opens, validates and initializes all restore
434    *   plug-in (shared) libraries.  They must be located in
435    *   /usr/epoch/EB/cure_plugin ( eb_cure_plugin_dir).  All .so files in that
436    *   directory are opened and validates for version# and presence of all
437    *   mandatory functions.  The RSTPI_Identify function is called for each
438    *   library to determine which optional features are supported, and that
439    *   the corresponding functions are present.  Finally, the RSTPI_Initialize
440    *   function is called for each valid library.
441    *
442    *   Parameters:
443    *
444    *   Inputs:
445    *
446    *     rcp      (I)    - Pointer to restore context
447    *
448    *   Outputs:
449    *
450    *     none
451    *
452    *   Returns:
453    *
454    *     E_SUCCESS or EP_RB_RECOVER_xxx
455    *
456    *   Logic/pseudo code:
457    *
458    *   open plugin dir
459    *   while read_next_entry succeeds
460    *     verify .so file (else continue)
461    *     open shared library file (else continue)
462    *       on errors below:
463    *         close shared library file
464    *         continue
465    *       fetch all mandatory function addresses
466    *       call Identify function
467    *       validate version number
468    *       fetch all indicated optional function addrs
469    *       call Initialize function
470    *       add worktiem types to composite exclusion list
471    *       add to valid plugin list
472    *   close plugin dir
473    *   */
474  1 static eerrno_ty init_plugins( restore_context *rcp )
471  1 {
472  1     DIR                 *dirp;
473  1     struct dirent       *direntp;
474  1     eerrno_ty            status = E_SUCCESS;
475  1     struct pluginData   *piDataPtr = NULL;
476  1     struct pluginData   *piListPtr = NULL;
477  1     int                  val_result;
478  1     struct pluginIDdata *idDataPtr;
479  1     char                *tmp_types;
480  1     int                  shlib_dirlen;
481  1     char                 shlib_path [MAXPATHLEN];
482  1
```

```
484  1 #if 1
485  1     /* open plugin directory or bust */
486  1     if ( NULL == (dirp = opendir( eb_cure_plugin_dir ))) )
487  2     {
488  2        rec_api_log_csm( SUB_CSM_PLUGIN_ERR, NULL );
489  2
490  2        return E_SUCCESS;            /* allow continuation w/o plugins */
491  2     }
492  2 #else
493  1     return EP_RB_RECOVER_NO_PLUGINS;          /* later do this */
     #endif

495  1     strcpy( shlib_path, eb_cure_plugin_dir );
496  1     strcat( shlib_path, "/" );
497  1     shlib_dirlen = strlen (shlib_path);
499  1
500  1     /* loop thru entries in directory */
501  2     while (NULL != (direntp = readdir( dirp )) )
502  2     {
503  2        if (NULL == piDataPtr)
504  3        {  /* allocate next plugin data structure */
505  3           if (NULL == (piDataPtr
506  4                = calloc( 1, sizeof(
507  4                    struct pluginData) )))
508  4           {
509  3              status = EP_RB_RECOVER_NOMEM;
510  2              break;           /* fall thru to cleanup */
512  2           }
513  2        }
515  2        if (NULL == strstr( direntp->d_name, ".so" ) )
516  2           continue;           /* skip this guy */
517  2
518  2        strcpy( &shlib_path[shlib_dirlen], direntp->d_name );
519  3        if (NULL == (piDataPtr->libHdl
520  3             = dlopen( shlib_path, RTLD_NOW )))
521  3        {
522  2           rbe_user_error( 0,
523  2              "Error opening restore plug-in library %s: %s\n",
525  2              direntp->d_name, dlerror() );
526  2           continue;           /* skip this one */
528  2        }
529  2
     /* Fetch addresses of all mandatory functions and */
     /* Do Identify processing: call it, save options, validate */
530  2        if (0 != (val_result = validate_plugin(
531  4           piDataPtr ) ))
532  4        {
533  4           if (val_result == -1 || val_result == -4)
534  3           {
535  3              rbe_user_error( 0,
536  3                 "Functions missing from restore plug-in library %s:\n",
537  4                 direntp->d_name, dlerror() );
538  4           }
539  4           else if (val_result < 0)
540  4           {
541  3              rbe_user_error( 0,
                       "Validation failed for restore plug-in
                        library %s\n",
                       direntp->d_name );
```

```
542  3          else
543  4          {
544  4              rbe_user_error( val_result,
545  4                  "RSTPI_Identify failed for restore plug-in library
546  4                  %s\n",
547  3                  direntp->d_name );

549  3              dlclose( piDataPtr->libHdl );       /* close .so on errors */
550  3              piDataPtr->libHdl = NULL;
551  3              continue;                           /* on any error, skip this lib */
552  3          }

554  2
555  2          /* let DC plug-in do its initialization */
556  2          rcp->appData = NULL;                    /* enter plugin with clean appdata */
557  2          status =
558  3              piDataPtr->piFuncArray[PIFuncIndexInitialize]( rcp );
559  3          if (E_SUCCESS != status)
560  3          {
561  3              rbe_user_error( status,
562  3                  "RSTPI_Initialize failed for restore plug-in library
563  3                  %s\n",
564  3                  direntp->d_name );
565  3              dlclose( piDataPtr->libHdl );       /* close .so on errors */

566  2              piDataPtr->libHdl = NULL;
568  2              status = E_SUCCESS;                 /* this wasn't fatal */
569  2              continue;                           /* on any error, skip this lib */
570  2          }

572  2          /* save plugin's appData */
573  2          piDataPtr->appData = rcp->appData;
574  2          rcp->appData = NULL;
575  2
576  2          /* add piDataPtr to valid plugin list */
577  2          if (NULL == piListPtr)
578  2              rcp->piList= piDataPtr;             /* first in list */

580  2          else
581  2              piListPtr->next = piDataPtr;        /* link from prev */
582  2          piListPtr = piDataPtr;                  /* new end of list */
583  3          piDataPtr = NULL;
584  3
585  3          /* add workitem types to composite exclusion list */
586  4          idDataPtr = (struct pluginIdData *)piListPtr->idData;
587  4          if (idDataPtr->num_types > 0)
588  4          {
589  3              tmp_types = calloc( 1, 1 + idDataPtr->num_types
590  3                  + rcp->rc_num_plugin_wi_types
591  4                  );
592  4              if (NULL == tmp_types) {
593  4                  status = EP_RB_RECOVER_NOMEM;
594  4                  break;
```

```
595  4                  }
596  3                  if (NULL != rcp->rc_plugin_wi_types)
597  3                  {
598  3                      /* move old list to new buffer and free old list */
599  3                      memcpy( tmp_types,
600  3                          rcp->rc_plugin_wi_types,
601  3                          rcp->rc_num_plugin_wi_types );
602  3                      free( rcp->rc_plugin_wi_types );
603  2                  }
604  1                  memcpy( tmp_types + rcp->rc_num_plugin_wi_types,
606  1                      idDataPtr->wi_types,
608  1                      idDataPtr->num_types );
609  1                  rcp->rc_num_plugin_wi_types +=
610  1                      idDataPtr->num_types;
612  1                  tmp_types[rcp->rc_num_plugin_wi_types] = 0;
613  2                  rcp->rc_plugin_wi_types = tmp_types;
614  2              }
615  3          }
616  3
617  3          (void)closedir( dirp );
618  3
619  3          /* free up leftovers: */
620  2          if (NULL != piDataPtr)
621  2              free (piDataPtr);

622  1          if (E_SUCCESS != status)
624  1          {   /* Free contents of plugin list: */
625  1              while (NULL != (piDataPtr = piListPtr))
                    {   /* allow plugin to clean up and close .so: */
                        rcp->appData = piDataPtr->appData;
                        piDataPtr->piFuncArray[PIFuncIndexFinish](
                            rcp );
                        dlclose( piDataPtr->libHdl );
                        piListPtr = piDataPtr->next;
                        free (piDataPtr);
                    }
                }

                return status;
            }
```

```c
                                      /* init_plugin */

/* ***********************************************************
 *
 *    validate_plugin
 *
 *    Function Description:
 *
 *    This function retrieves the addresses of the mandatory plugin
 *    functions
 *    and stores them in the function pointer array.
 *                                         If any function is missing
 *    it returns -1.
 *    It then calls the identify function and verifies wthe plugin
 *    version,
 *        and finds its optional functions. Specific error values are
 *    returned on version mismatch and missing optional functions.
 *
 *    Parameters:
 *
 *    Inputs:
 *        piDataPtr
 *            I)      - Pointer to plugin data structure with libHdl set
 *
 *    Outputs:
 *        piFuncArray in piDataPtr is loaded with pointers to plugin
 *                                                        functions
 *
 *    Returns:
 *        0 on success
 *        -1 on any missing required functions
 *        -2 if version validation fails OR identify returns junk
 *        -3 if workitem type validation fails
 *        -4 on any missing optional functions indicated by options
 *                                                     flags
 *        +n (
 *    EB_RB_RECOVER_xxx) for error codes returned from Identify function
 *
 * ****************************************/

static int validate_plugin( struct pluginData *piDataPtr )
{
    int                 index;
    eerrno_ty           status;
    struct pluginData  *idDataPtr;

    for( index = 0; index <= PIFuncIndexLastBasic; index++ )
    {
        if (NULL == (piDataPtr->piFuncArray[index]
                = (piFuncPtr) dlsym( piDataPtr->libHdl,
                piFuncNames[index]
                )))
        {
            return -1;
        }
    }

    /* call identify and validate: */
    status = piDataPtr->piFuncArray[PIFuncIndexIdentify](
                &piDataPtr->idData );

    if (status != E_SUCCESS)
        return status;
    if (NULL == (idDataPtr =
                struct pluginIdData *)piDataPtr->idData)
        return -2;
```

```c
    if (idDataPtr->version != RSTPI_VERSION)
    { /* only version 1 supported so far */
        piDataPtr->idData = NULL;
        return -2;
    }

    if ( ( idDataPtr->num_types && !idDataPtr->wi_types)
            /* count cant be positive with null pointer */
        && (NULL ==
            piDataPtr->idData = NULL;
            return -3;
    }

    /* if startRestore option set, get its addr or bust */
    if ( ( (idDataPtr->options & RSTPI_OPTION_MASK_START)
        == (RSTPI_OPTION_SPECIAL_START)
        && ( (NULL == (piDataPtr->piFuncArray[PIFuncIndexStartRestore]
                = (piFuncPtr) dlsym( piDataPtr->libHdl,
                piFuncNames[PIFuncIndexStartRestore] )))

    /* OR if special find option set, get its addr or bust */
        || ( (idDataPtr->options & RSTPI_OPTION_SPECIAL_FIND)
        == (idDataPtr->options & RSTPI_OPTION_MASK_FIND))
        && ( (NULL == (piDataPtr->piFuncArray[PIFuncIndexFindResults]
                = (piFuncPtr) dlsym( piDataPtr->libHdl,
                piFuncNames[PIFuncIndexFindResults]

        || ( NULL == (
                piDataPtr->piFuncArray[PIFuncIndexFind]
                = (piFuncPtr) dlsym( piDataPtr->libHdl,
                piFuncNames[PIFuncIndexFind] ) ) )
        )))

    /* OR if special getMedia option set, get its addr or bust */
        || ( (idDataPtr->options & RSTPI_OPTION_MASK_GET_MEDIA)
        ==
        ( (RSTPI_OPTION_SPECIAL_GET_MEDIA)
        && (NULL == (piDataPtr->piFuncArray[PIFuncIndexGetMedia]
                = (piFuncPtr) dlsym( piDataPtr->libHdl,
                piFuncNames[PIFuncIndexGetMedia] )))
    {
        piDataPtr->idData = NULL;
        return -4;
    }

    return 0;
}
```

723

```
/*                    validate_plugin */
```

```
1   #define __POSIX_SOURCE 1

3   #include <restore/restoretree.h>
4   #include <restore/RSLplugin.h>
5   #include <ebconfig/rbconfig.h>

8   //
```

```
10       extern "C" void dump_unix_time(RSTRPC_time_ty time)
11  1    {
12  1        DateTime t(time);
13  1        if(time)
14  2        {
15  2            cout << "Time: " << t << endl ;
16  1        }
17  1        else
18  2        {
19  2            cout << "Time: Zero value\n";
20  1        }
21  1    }
```

```
23      extern "C" void dump_time_list(RSTRPC_time_list *list, ostream &out)
24 1    {
25 1        out << "Dump of time list\n";
26 1        DateTime *t;
28 1        for(RSTRPC_time_list *listelem=list;listelem;
                        listelem=listelem->next)
29 2        {
30 2            t=new DateTime(listelem->time);
31 2            out << " Time: " << *t << endl;
32 2            free (t);
33 1        }
35 1        out << "End of time list\n";
37      }
```

```
39      extern "C" void dump_tlo_list(RSTRPC_tlo_list *list,ostream &out)
40 1    {
41 1        out << "Dump of RSTRPC_tlo_list\n";
42 1        for(;list;list=list->next)
43 2        {
44 2            RSTRPC_top_level_obj *tobj=list->tlo;
45 2            RSTRPC_restorable_obj_root *ro=&(tobj->root);
46 2            out << "TLO " << endl;
47 2            out << "   level:" << ro->objLevel << "   Backup App:"<<
                        ro->backupApp<<endl;
48 2            if(ro->objName)
49 3            {
50 3                out << "   name :" << ro->objName <<endl;
51 2            }
52 2            if(ro->objTypeString)
53 3            {
54 3                out << "   types:" << ro->objTypeString << endl;
55 2            }
56 1        }
57 1        out << "End of dump RSTRPC_tlo_list\n";
59 1        return;
60      }
```

```
62
```

`//`

```
63     extern "C" void dump_uro_list(RSTRPC_uro_list *list,ostream &out)
64  1  {
65  1      out << "Dump of RSTRPC_uro_list\n";
66  1      for(;list;list->next)
67  2      {
68  2          out << "We have a node\n";
69  1      }
70  1      out << "End of dump RSTRPC_uro_list\n";
71      }
```

73

//

```
74      eerrno_ty RSTPI_Initialize(restore_context *context)
75    1 {

77    1     // See if an initial sleep is needed
78    1     /* char *sl=getenv("RSTPI_INITIALIZE_SLEEP");
79    1     if(!sl)
80    1     {
81    1         int st=atoi(sl);
82    1         rbe_log_stats(0,"RSTPI_Initialize, sleep of %d seconds",st);
83    1         sleep(st);
84    1     }
85    1     else
86    1     {
87    1         rbe_log_stats(0,"RSTPI_Initialize, no delay on startup");
88    1     }
89    1     */
90    1     // Allocate restore context data
91    1     context->appData=(void *)malloc(sizeof(struct restoreContextData));

93    1     if(!context->appData)
94    2     {
95    2         rbe_log_stats(0,"plugin.cc - malloc failure");
96    2         return EP_RB_RECOVER_MALLOC_FAILURE;
97    1     }

99    1     struct restoreContextData *rcd=
100   1         (struct restoreContextData *)(context->appData);

102   1     rcd->currentWisetListNode=NULL;
103   1     rcd->currentWisetNode=NULL;
104   1     rcd->currentBackupNode=NULL;

106   1     return E_SUCCESS;
107   1 }
```

```
111   /********************************************************
112    *
113    * Submit
114    *
115    * This function creates a submit object from the currently marked
116    * restorable objects.  The ID of the created submit object is passed to
117    * EDMRST_Start to begin execution of the restore.
118    *
119    * Parameters:
120    *
121    *   context    (I) - Pointer to the restore context
122    *   hostName   (I) - host to restore to (only if inPlace == False)
123    *   policy     (I) - The overwrite policy to use
124    *   inPlace    (I) - Flag if the restoral is to be in original locations
125    *   directory  (I) - directory to restore to (
126    *                          only if inPlace == False)
127    *   submitObjIDptr  (IO) - ID of the submit user object created to describe
128    *                          the restore
129    *   progressCB (I) - pointer to callback function to report progress and
130    *                          test for cancellation
131    *   transport  (I) - Indicator of transport the restoral is to be over (SCSI
                                    or network)
132    *
133    *   Note:
134    *              The Progress Callback is currently not used.  If the need
135    *              for this developes, the routines which use this argument
136    *              (mark, unmark, submit) must be enhanced.
137    *
138    ********************************************************/
139
140   eerrno_ty  RSTPI_Submit( const restore_context       *context,
141                            const char                  *hostName,
142                            const OverwritePolicy        policy,
143                            const boolean_ty             inPlace,
144                            const char                  *directory,
145                            const RestoreTransport       transport,
146                            int                         *submitObjIDptr,
147                            RSTPI_SubmitProgressProc     progressCB )
148   {
149       // Get the restore context so we can find the app data
150       //
151       struct restoreContextData *rcd=(struct restoreContextData *)(
                                            context->appData);
153       if(!rcd)
154       {
155           return EP_RB_RECOVER_RC_APP_DATA_NULL;
156       }
158
159       // Get the current backup node from the top level object
160       //
161       if(!rcd->currentBackupNode)
162       {
163           return EP_RB_RECOVER_CURRENT_BACKUP_NODE_NULL;
164       }
167       BackupNode *cbu=rcd->currentBackupNode;
169       bool scode=cbu->fillSubmitObject(context,
```

```
170                                        hostName,
171                                        policy,
172                                        inPlace,
173                                        directory,
174                                        transport,
175                                        submitObjIDptr,
176                                        progressCB);
177       if(!scode)
178       {
179           rbe_log_stats(0,"RSTPI_Submit - error from fillSubmitObject");
180           return EP_RB_RECOVER_FILLSUBMIT_ERROR;
181       }
184       return E_SUCCESS;
186   }
```

```c
190  /**************************************************************************
191   *
192   * Get Top Level Objects
193   *
194   * This function is called to retrieve the configurable backup objects (work
195   * items for network backups and work item sets for Symmetrix Connect
196   *                                                                 backups,
197   * which are restorable for the given client.
198   * It is a GOAL of this routine to return all objects ever backed
199   * up successfully. For network backups, though,
200   *                                              it only looks in the config
201   * file for 'top level objects' of the given client.
202   *
203   * While the restore API will be called repeatedly to retrieve a
204   *                                                            maximum
205   * number of items on each call,
206   *                              this plug-in call must retrive the whole
207   * set of applicable backup objects.
208   *                                  The generic restore service library
209   * will manage the composite list of top level objects from all
210   *                                                       backup apps.
211   *
212   ***************************************************************************/
213   *
214   * Parameters:
215   * context       (I) - Pointer to the restore context
216   * sourceHost    (I) - the name of the host whose backups are being restored
       * topLevelObjs  (O) - ptr to linked list of Top Level Objects
       * numberEntries (O) - the real number of objects returned in the list
       *
       * Returns:
       *   E_SUCCESS            on success
       *   EP_RB_RECOVER_xxx    on error
       *
       ***************************************************************************/
218  eerrno_ty RSTPI_GetTopLevelObjects(restore_context *context,
219                                     const char * sourceHost,
220                                     struct RSTRPC_tlo_list **topLevelObjs,
221                                     short *numberEntries)
222  {
223      //
224      // Now look through current backups and put on list if in
225      // Range
226      //
227      struct restoreContextData *rcd=(struct restoreContextData *)(
                                                         context->appData);
229      if(!rcd)
230      {
231          return EP_RB_RECOVER_RC_APP_DATA_NULL;
232      }

235      rcd->currentWisetListNode=new WiSetListNode((char *)sourceHost);

238      //
239      // Populate this node only
240      //
241      if(!rcd->currentWisetListNode->populate(
```

```c
                                     context, POPULATE_NODE, sourceHost))
242      {
243
244          return EP_RB_RECOVER_WISETLIST_NODE_POPULATE_ERR;
246      }

247      *topLevelObjs=NULL;
         *numberEntries=0;
249      RSTRPC_tlo_list *previous=NULL;

252      for(RestoreNode *rn  =rcd->currentWisetListNode->getFirstChild();
253          rn; rn=rcd->currentWisetListNode->getNextChild())
254      {
255          //
256          (*numberEntries)++;
257          //
258          RSTRPC_tlo_list *new_list_elem=(RSTRPC_tlo_list *)malloc(sizeof(
                                                   struct RSTRPC_tlo_list));
259          if(!new_list_elem)
260          {
261              rbe_log_stats(0, "plugin.cc - malloc failure");
262              return EP_RB_RECOVER_MALLOC_FAILURE;
263          }
264          RSTRPC_top_level_obj *new_top=(RSTRPC_top_level_obj *)malloc(
                                         sizeof(struct RSTRPC_top_level_obj));
265          if(!new_top)
266          {
267              rbe_log_stats(0, "plugin.cc - malloc failure");
268              return EP_RB_RECOVER_MALLOC_FAILURE;
269          }
270          memset(new_top, 0, sizeof(struct RSTRPC_top_level_obj));

273          /*
274           * We need to check to see if a network is restorable for this
275           * top level object. Since the client machines are the same for
276           * all backups of the same workitem (duh!) we only need to check
277           * one. Although this should always work, we will behave in a
278           * failsave manner by
279           *
280           * (a) Marking sure the child is non-null
281           *
282           * (b) Marking network as "not possible" when we find that case
283           *     (so we consider network restore possible, and an allowed
284           *     option, if we don't find a client in the ddtab for the
285           *     first backup of the work item)
286           */
287          BackupNode *bn=(BackupNode *)rn->getFirstChild();
288          // C++ standard evaluates left to right, so this is null-safe
289          if(bn && !bn->getNetworkRestorePossible())
290          {
291              new_top->flags |= TLO_BITFLAG_NETWORK_RESTORE_NOT_POSSIBLE;
292          }

294          char ** temp_ptr;
295          temp_ptr=(char **)malloc(sizeof(char *));
296          if(!temp_ptr)
297          {
298              rbe_log_stats(0, "plugin.cc - malloc failure");
299              return EP_RB_RECOVER_MALLOC_FAILURE;
300          }
301          *temp_ptr=(char *)rn;
302          new_top->appData.data=(char **)temp_ptr;
303          new_top->appData.length=sizeof(char *);
```

```
305  2    new_top->root.objLevel=RSTRPC_tlo_type;
306  2    new_top->root.objName=esl_strdup((char *)(rn->getNameChar()));
307  2    if(NULL==new_top->root.objName)
308  3    {
309  3       rbe_log_stats(
310  3          0,"plugin.cc - allocation failure from esl_strdup failure");
311  2       return EP_RB_RECOVER_MALLOC_FAILURE;
         }

313  2    new_top->wiBIC=NULL;
314  2    //
315  2    new_top->templateName=esl_strdup("");  // Blank as placeholder

317  2    new_top->objTypeString=esl_strdup("");
                                              // Blank as placeholder
318  2    new_top->hostname=esl_strdup(rcd->currentWisetListNode->getName(
                                                            ));
319  2    new_top->fileSpec=esl_strdup("");  // Blank as placeholder

321  2    if(NULL == new_top->root.objTypeString ||
322  2       NULL == new_top->hostname ||
323  2       NULL == new_top->fileSpec)
324  3    {
325  3       rbe_log_stats(
326  3          0,"plugin.cc - allocation failure from esl_strdup");
327  2       return EP_RB_RECOVER_MALLOC_FAILURE;
         }

329  2    new_top->wiType='\0';
330  2    new_top->ssThread=FALSE;

332  2    new_list_elem->tlo=new_top;
333  2    new_list_elem->next=NULL;

335  2    if(previous)
336  3    {
337  3       previous->next=new_list_elem;
338  2    }
339  2    else
340  3    {
341  3       *topLevelObjs=new_list_elem;
342  2    }
343  2    previous=new_list_elem;

345  1    }

347  1    return E_SUCCESS;
348  1  }
```

```
350  //
```

```
352  /****************************************************************
353  *  Set TopLevel Object:
354  *
355  *  This function is called to notify the plug-in that one of its top
                level
357  *  objects has been selected (for browsing and marking). The plug-in
358  *  should perform whatever validation and initialization is needed to
                prepare
359  *  for browsing and marking this top level object.
360  *  If this call returns success,
                                    then RSTPI_GetNextLevelObjects will be
361  *  called to retrieve the highest level restorable objects for this
362  *  backup object.
363  *  NOTE: This function is responsible for setting the template_name
                and
364  *  saveset_thread elements of the restore context.
365  *
366  *  Returns:
367  *      E_SUCCESS            on success
368  *      EP_RB_RECOVER_xxx    on error
369  *
370  *  Parameters:
371  *      context       (I)  -  Pointer to the restore context
372  *      backupObject  (I)  -  the selected top level object
373  *
374  ****************************************************************/
375
376
377  eerrno_ty RSTPI_SetTopLevelObject(restore_context *context,
                                       struct RSTRPC_top_level_obj
                                              *backupobject)
381  {
        struct restoreContextData *rcd=(struct restoreContextData *)(
                                         context->appData);
        //
        // Get the restore context so we can find the app data
        //
383     if(!rcd)
384     {
385         return EP_RB_RECOVER_RC_APP_DATA_NULL;
386     }

388     char ** c1=(char **)(backupobject->appData.data);
389     //
390     // Get the current node info from the top level object
391     //
393     RestoreNode *rnodep=(RestoreNode *)(*c1);
395     if(!rnodep)
396     {
397         return EP_RB_RECOVER_RN_APP_DATA_NULL;
398     }
400     //
401     // Top level object should always point to a work item set node
402     // is the case.
403     //
404     if(rnodep->nodeType() != RNC_WI_SET)
405     {
406         return EP_RB_RECOVER_TLO_NOT_WISET_NODE;
407     }
409     // Set the current backup node
410
```

```
411     //
412     rcd->currentWisetNode=(WiSetNode *)rnodep;
414     //
415     // Set the current backup node to the most recent one in time
416     // which is FULL
417     //
419     rcd->currentBackupNode=rnodep->getNextBackupNode
420     if(NULL != rcd->currentBackupNode)
421     {
422         rcd->currentBackupNode->unlockWorkitems(context);
423         rcd->currentBackupNode->unmarkNode(TRUE,FALSE);
424     }
425     rcd->currentBackupNode=NULL;

427     // Clear locks if they exist
428     BackupNode *bnsave=NULL;
429     BackupNode *nextBnode;
        int more=1;
431     for(BackupNode *bn=(BackupNode *)rnodep->getFirstChild();
432         more;
433         bn=nextBnode)
434     {
435         nextBnode=(BackupNode *)(rnodep->getNextChild());
437         if(NULL == nextBnode) // Force end of loop if no more items to
                                     scan
438         {
439             more=0;
440         }

442         //
443         // Populate in case this is necessary
444         //
445         // Errors will set appropriate state bits on nodes
446         int err=bn->populate(context, POPULATE_CHILDREN);
448         //
449         // If we already have a current backup node,
                                                     clear all marks
450         if(bn->getstateBit(STATE_COMPLETE))
451         {
452             bnsave=bn;  // Save in case we have to return an incomplete one
453             if(rcd->currentBackupNode)
454             {
455                 rcd->currentBackupNode->unlockWorkitems(context);
456                 rcd->currentBackupNode->unmarkNode(TRUE,FALSE);
457             }
458             rcd->currentBackupNode=bn;
459             if(!rcd->currentBackupNode->lockworkitems(context))
460             {
461                 return EP_RB_RECOVER_WI_LOCKED;
462             }
463             RnProperty *tpnode=bn->getProperty(PROPERTY_TEMPLATE);
464             if(!tpnode)
465             {
466                 return EP_RB_RECOVER_NO_TEMPLATE_PROPERTY;
467             }
468             char * templateBase=(char *)(((
                          RnPropertyChar *)tpnode)->getValue());
469             backupobject->templateName=esl_strdup(templateBase);
                context->rc_template_name=esl_strdup(templateBase);
471             if(NULL == backupobject->templateName ||
472                 NULL == context->rc_template_name)
473             {
```

```
474  4        rbe_log_stats(
475  4            0, "plugin.cc - allocation failure from es1_strdup");
476  3        return EP_RB_RECOVER_MALLOC_FAILURE;
478  3    }
479  3
480  3    // Break out so that we get the first one on the list
481       // (most recent)
482  2    more=0;
484  1    }
486  1    if(!rcd->currentBackupNode)
487  1    {
488  1        //
489  1        // Check and make sure we got one
490  2        //
491  2        if(NULL==bnsave)
492  2        {
493  3            // We shouldn't be able to get here,
494  3            // but let's check for safety
495  3            rbe_log_stats(0, "Could not find any backup node");
496  2            return EP_RB_RECOVER_NONE_EXISTS;
498  2        }
499  3        rcd->currentBackupNode=bnsave;
500  3        rcd->currentBackupNode->unlockWorkitems(context);
501  2    }
502  2    if(!rcd->currentBackupNode)
503  2    {
504  3        return EP_RB_RECOVER_WI_LOCKED;
505  3    }
506  2    }
507  2    RnProperty *tpnode=rcd->currentBackupNode->getProperty(
                          PROPERTY_TEMPLATE);
508  2    if(!tpnode)
509  3    {
510  3        return EP_RB_RECOVER_NO_COMPLETE_BACKUP;
511  2    }
512  2    char * templateBase=(char *)((
                          RnPropertyChar *)tpnode)->getValue());
513  2    backupobject->templateName=es1_strdup(templateBase);
514  2    context->rc_template_name=es1_strdup(templateBase);
516  2    if(NULL==backupobject->templateName ||
517  2       NULL==context->rc_template_name)
518  3    {
519  3        rbe_log_stats(
520  3            0, "plugin.cc - allocation failure from es1_strdup");
521  2        return EP_RB_RECOVER_MALLOC_FAILURE;
522  1    }
524  1    return E_SUCCESS;
525  1    }
```

```
527      //
```

```
528  2   /*****************************************************************
529  2   *
530  2   * Get Next Level Objects:
531      *
532      * This function is intended to allow retrieval of the children
533      * of a given parent object.
534      *
535      * The caller specifies the parent object and
536      * whether or not to include bad files. Even though the objects are
537      * returned in a linked list, there could conceivably be thousands of
538      * child objects, so the caller must specify the maximum number
539      * of children to return. The caller is returned a token (
540      * cookie) to allow
541      * continuing on the next call to thid function.
542      *
543      * Parameters:
544      *
545      *    context        (I) - Pointer to the restore context
546      *    parentObject   (I) - the parent object
547      *    objectLevel    (I) - specifies whether parentObject is a top level or
548      *                         container object
549      *
550      *    objects        (O) - a pointer to receive the start of the objects list
551      *    cookie         (IO) - a place holder for the list position
552      *    maxEntries     (I) - the maximum number of objects to return
553      *    numberEntries  (O) - the real number of objects returned in the list
554      *    allowBadFiles  (I) - flag whether or not to include bad files
556      * Returns:
557      *    E_SUCCESS             on success
558      *    EP_RB_RECOVER_xxx     on error
559      *
560      *****************************************************************/
561      eerrno_ty RSTPI_GetNextLevelObjects(restore_context *context,
562                                          restorableObjectPtr parentObj,
563                                          enum RSTRPC_ObjectLevel
564                                              objectLevel,
                                             struct RSTRPC_uro_list **objects,
                                             long *cookie,
                                             const long maxEntries,
                                             long *numberEntries,
                                             const boolean_ty allowBadFiles)
564  1   {
566  1   struct restoreContextData *rcd=
567  1      (struct restoreContextData *)(context->appData);
569  1   if(!rcd)
570  2   {
571  2      return EP_RB_RECOVER_RC_APP_DATA_NULL;
572  1   }
574  1   //
575  1   // Geta the restore node of the child
576  1   //
577  1   RestoreNode *rnodep;
579  1   char ** cl;
581  1   switch(objectLevel)
582  2   {
583  2   case RSTRPC_tlo_type:
584  2      cl=(char **)(((RSTRPC_top_level_obj *)parentObj)->appData.data);
585  2      rnodep=(RestoreNode *)(*cl);
586  2      break;
587  2   case RSTRPC_container_type:
```

```
588  2   case RSTRPC_leaf_type:
589  2      cl=(char **)(((
590         RSTRPC_user_restorable_object *)parentObj)->appData.data);
591  2      rnodep=(RestoreNode *)(*cl);
592  2      break;
593  2   default:
594  2      return EP_RB_RECOVER_BAD_OBJECT_LEVEL;
595  1      break;
         }
597  1   if(!rnodep)
598  2   {
599  2      return EP_RB_RECOVER_RN_APP_DATA_NULL;
600  1   }
602  1   //
603  1   // If we are getting the children of the current workitem set node, we
604  1   // acrually want the children of the current backup node. The current backup node
605  1   // is the one which pertains to the point in time we are interested in.
606  1   //
608  1   if(rnodep==rcd->currentTwisetNode)
609  2   {
610  2      rnodep=rcd->currentBackupNode;
611  1   }
613  1   if(!rnodep)
614  2   {
615  2      return EP_RB_RECOVER_CURRENT_BACKUP_NODE_NULL;
616  1   }
619  1   int entries_count=0;
621  1   //
622  2   // This code checks to see if we have a cookie.  If the cookie is non-null,
623  1   // it is actually a pointer to the position int he chain where we left
624  1   // off.
625  1   //
626  1   // Danger Will Robinson - this relies on ebing able to fit a pointer
627  1   // into a cookie (long int).
628  1   //
629  1   RestoreNode *child;
630  1   if(0 == *cookie)
631  2   {
632  2      child=rnodep->getFirstChild();
633  1   }
634  1   else
635  2   {
636  2      // Standards issue: Caution,
637  2                 assuming long can be cast to/from pointer
638  1      child=(RestoreNode *)*cookie;
         }
640  1   RSTRPC_uro_list *previous=NULL;
641  1   RSTRPC_uro_list *listentry;
642  1   RSTRPC_user_restorable_object *urobj;
644  1   //
645  1   // Traverse the children of the current node and create a restorable
646  1   // object list.
```

```
647  1  //
648  1  for(;child;child=rnodep->getNextChild())
649  1  {
650  2      //
651  2      // Skip this entry if we have chosen not to allow bad files
652  2      //
653  2      if( (FALSE == allowBadFiles) &&
654  2          (0 != rnodep->getStateBit(STATE_BADFILE)) )
655  3      {
656  3          continue;
657  2      }

659  2      entries_count++;

661  2      listentry=(RSTRPC_uro_list *)malloc(sizeof(
                                      struct RSTRPC_uro_list));
662  2      if(!listentry)
663  3      {
664  3          rbe_log_stats(0,"plugin.cc - malloc failure");
665  3          return EP_RB_RECOVER_MALLOC_FAILURE;
666  2      }

668  2      urobj=(RSTRPC_user_restorable_object *)malloc(sizeof(
                                      struct RSTRPC_user_restorable_object));
669  2      if(!urobj)
670  2      {
671  3          rbe_log_stats(0,"plugin.cc - malloc failure");
672  3          return EP_RB_RECOVER_MALLOC_FAILURE;
673  3      }

674  3      memset(urobj,0,sizeof(RSTRPC_user_restorable_object));

676  2      listentry->uro=urobj;

678  2      char ** temp_ptr;
679  2      temp_ptr=(char **)malloc(sizeof(char *));
680  2      if(!temp_ptr)
681  3      {
682  3          rbe_log_stats(0,"plugin.cc - malloc failure");
683  3          return EP_RB_RECOVER_MALLOC_FAILURE;
684  2      }
685  2      *temp_ptr=(char *)child;

687  2      urobj->appData.data=(char *)temp_ptr;
688  2      urobj->appData.length=sizeof(char *);

690  2      urobj->root.objLevel=child->getStateBit(
                STATE_LEAFNODE)?RSTRPC_leaf_type:RSTRPC_container_type;

692  2      char *child_name=(char *)(child->getNameChar());
693  2      urobj->root.objName=esl_strdup(child_name);

695  2      if(NULL==urobj->root.objName)
696  3      {
697  3          rbe_log_stats(
698  3              0,"plugin.cc - allocation failure from esl_strdup");
699  2          return EP_RB_RECOVER_MALLOC_FAILURE;
            }

701  2      // Find base name - token after final '/' if present,
702  2      // else whole thing
703  2      // If the final character is a "/", look for the token preceding
         2      // this final character.

705  2      char *rp=NULL;
706  2      char *ch=NULL;
707  2      RnProperty *gname=child->getProperty(PROPERTY_GUINAME);
```

```
708  2      if(gname)
709  3      {
710  3          rp=(char *)((RnPropertyChar *)gname)->getValue();
711  2      }

713  2      if(NULL==rp)
714  3      {
715  3          ch=esl_strdup(child_name);
716  3          if(NULL==ch)
717  4          {
718  4              rbe_log_stats(
719  4                  0,"plugin.cc - allocation failure from esl_strdup");
720  3              return EP_RB_RECOVER_MALLOC_FAILURE;
721  3          }
722  3          int len=strlen(child_name);
723  3          if(ch[len-1] == '/' && len>1)
724  4          {
725  3              ch[len-1]='\0';
            }

727  3          rp=rindex(ch,'/');

729  3          if(NULL==rp)
730  4          {
731  4              rp=(char *)(child->getNameChar());
732  3          }
733  3          else
734  4          {
735  4              rp++;
736  3          }
737  3      }

738  2      urobj->root.objTypeString=NULL; // ROB: We neeed to fill this in
739  2      if(NULL!=ch)
740  3      {
741  3          free(ch);
742  2      }

744  2      urobj->root.objBaseName=esl_strdup(rp);

746  2      char *tmp_owner=uid2str(child->getOwner());
747  2      char *tmp_group=gid2str(child->getGroup());

749  2      if(NULL==tmp_owner || NULL==tmp_group)
750  3      {
751  3          rbe_log_stats(
752  2              0,"plugin.cc - allocation failure from esl_strdup");
753  2          return EP_RB_RECOVER_MALLOC_FAILURE;
            }

755  2      urobj->objMode=child->getMode();

757  2      //
758  2      // Archive log file node stores owner and group as character
759  2      // strings since we got it from the Network restore API
760  2      //
761  2      if(child->nodeType() == RNC_ARCHIVELOGFILE)
762  3      {
763  3          ArchivelogfileNode *anode=(ArchivelogfileNode *)(child);
764  3          urobj->objOwnerName=esl_strdup(anode->getOwnerChar());
765  3          urobj->objGroupName=esl_strdup(anode->getGroupChar());
766  2      }
767  3      else
768  3      {
769  3          urobj->objOwnerName=esl_strdup(tmp_owner);
770  3          urobj->objGroupName=esl_strdup(tmp_group);
771  2      }
```

```
773  2      if(NULL==urobj->objOwnerName || NULL==urobj->objGroupName)
774  3          rbe_log_stats(
775  3              0,"plugin.cc - allocation failure from esl_strdup");
776  3          return EP_RB_RECOVER_MALLOC_FAILURE;
777  2      }


780  2      RnPropertyDate *pr=(RnPropertyDate *)(child->getProperty(
                                  PROPERTY_BACKUP_DATE));
781  2      if(pr)
782  3      {
783  3          urobj->objModTime=pr->getValue()->unixTime();
784  2      }
785  2      else
786  3      {
787  3          BackupNode *bnode=(BackupNode *)(child->getBackupNodePtr());
788  3          urobj->objModTime=0;
789  3          if(bnode)
790  4          {
791  4              pr=(RnPropertyDate *)(bnode->getProperty(
                                  PROPERTY_BACKUP_DATE));
792  4              if(pr)
793  5              {
794  5                  urobj->objModTime=pr->getValue()->unixTime();
795  4              }
796  3          }
797  2      }


799  2      urobj->objSize.high=child->getSize().high;
800  2      urobj->objSize.low=child->getSize().low;


802  2      if(child->getStateBit(STATE_BADFILE))
803  3      {
804  3          urobj->objBackupStatus=RSTRPC_Backup_Bad;
805  2      }
806  2      else if( child->getStateBit(STATE_INVALID_SSID |
807  2                   STATE_NO_SSID |
808  2                   STATE_EXPIRED_SSID ) )
809  3      {
810  3          urobj->objBackupStatus=RSTRPC_Backup_Expired;
811  2      }
812  2      else if( child->getStateBit(STATE_CH_INVALID_SSID |
813  2                   STATE_CH_NO_SSID |
814  2                   STATE_CH_EXPIRED_SSID |
815  2                   STATE_CH_BADFILE ) )
816  3      {
817  3          urobj->objBackupStatus=RSTRPC_Backup_Child_Without_Data;
818  2      }
819  2      else
820  3      {
821  3          urobj->objBackupStatus=RSTRPC_Backup_Good;
822  2      }

824  2      listentry->next=NULL;

826  2      if(previous)
827  3      {
828  3          previous->next=listentry;
829  2      }
830  3      else
831  3      {
832  3          *objects=listentry;
833  3      }
834  2      previous=listentry;
```

```
836  2          if(entries_count==maxEntries)
837  3          {
838  3              *cookie=(long)child;
839  3              *numberEntries=entries_count;
840  3              return E_SUCCESS;
841  2          }
842  1      }


844  1      *cookie=DONE_COOKIE;
845  1      *numberEntries=entries_count;
846  1      return E_SUCCESS;
847     }
```

```
849  //
```

```
851  /***********************************************************************************************
852   * Mark Object
853   *
854   * The MarkObject operation takes a restorableObject and marks it, and
855   *     possibly its descendant files for restoral based on the input
856   *     criteria.
857   * Since the RSTSL_MarkObject call is an asynchronously executed
858   *     operation
859   *     in the Restore Engine that performs the marking, this function must
860   *     periodically check for user-signalled cancelation, and update progress
861   *     data using the progress callback function argument.
862   * NOTE: This functions is responsible for keeping the volumes needed
863   *     list
864   *     (ebvllist) element of the restore context up to date.
865   *
866   *
867   * Parameters:
868   *
869   * context       (I) - Pointer to the restore context
870   * thisObject    (I) - The restoral object;
871   *                         can be a leaf object (e.g. a
872   *                         file), or a container object (
873   *                         e.g., a directory).
874   *
875   * allowBadfiles (I) - allows marking of files of state BADDATA.
876   * descend
877   *           I)  - Should mark operation descend to operate on the content
878   *                 of container objects.
879   * BadFilesCount (O)  - returns the file count with BADDATA.
880   * PermDenyFilesCount (
881   *           O) -- returns the file count with permission denied.
882   * fileMarked (
883   *           O) - return the total files marked after this mark occurred.
884   * lenMarkedFiles(
885   *           O) - return the length of files marked after this mark
886   * dirMarked
887   *           O) - return the total directories marked after this mark
888   *                 occurred.
889   * otherMarked (
890   *           O) - return the total "other" files marked after this mark.
891   * progressCB
892   *           I) - pointer to callback function to report progress and
893   *                 test for cancellation
894   *
895   ***********************************************************************************************/
896
897   Note:
898       The Progress Callback is currently not used.  If the need
899       for this developes, the routines which use this argument
         (mark, unmark, submit) must be enhanced.

eerrno_ty RSTPI_MarkObject(restore_context *context,
                           struct RSTRPC_user_restorable_object
                                                   *thisObject,
                           boolean_ty allowBadFiles,
                           boolean_ty descend,
                           unsigned long *badFilesCount,
                           unsigned long *permDenyFilesCount,
                           unsigned long *filesMarked,
                           u_hyper *lenMarkedFiles,
```

```
900                          unsigned long *dirMarked,
901                          unsigned long *otherMarked,
902                          RSTPI_MarkProgressProc progressCB)
903  1   {
904  1       char ** c1=(char **)(thisObject->appData.data);
905  1       RestoreNode *rnodep=(RestoreNode *)(*c1);

907  1       if(NULL==rnodep)
908  2       {
909  2           rbe_log_stats(
910  2               0,"RSTPI_MarkObject - Mark object has no app data");
911  1           return EP_RB_RECOVER_RN_APP_DATA_NULL;
             }

913         /*
914  1       * MTFB (multi trail file backup) nodes are to be treated as atomic.
915  1       * If the item we are trying to mark is a child of a MTFB node, then
916  1       * we perform the mark operation on the entire MTFB node.
917  1       */
918  1       RestoreNode *parent=rnodep->getParent();
919  1       if(NULL !=parent && parent->nodeType() == RNC_MTFB)
920  2       {
921  2           rnodep=parent;
922  1       }

924  1       *badFilesCount=0;
925  1       *permDenyFilesCount=0;
926  1       *filesMarked=0;
927  1       *lenMarkedFiles=ul_to_uh(0);
928  1       *dirMarked=0;
929  1       *otherMarked=0;

931  1       rnodep->markNode(descend,allowBadFiles);

933  1       // We start at the backup node pointer since we want the total
934  1       // for the entire backup, not just this node down (except the
935  1       // badFilesCount, in which case the affected nodes will only
936  1       // be from this level down)
938  1       *filesMarked=rnodep->getBackupNodePtr()->countMarkedNodes(
                                                RNC_ANY_DATAFILE);
939  1       *lenMarkedFiles=rnodep->getBackupNodePtr()->totalMarkedSize();

942  1       // The bad file count is a cound of the bad files encountered
943  1       // (not necessarily marked) from the current node down.
945  1       *badFilesCount=rnodep->countMarkedBadfileNodes(RNC_ANY_DATAFILE);

947  1       return E_SUCCESS;
948         }

950         //
```

```
/*************************************************************
 *
 * UnmarkObject
 *
 * The UnmarkObject operation takes a restorableObject and unmarks
 * possibly its descendant files for restoral based on the input
 * criteria.
 *
 * Since the RSTSL_UnmarkObject call is an asynchronously executed
 * operation
 *
 * in the Restore Engine that performs the unmarking,
 * this plug-in function
 *
 * must periodically check for user-signaled cancelation, and update
 * progress data using the progress callback function argument.
 *
 * NOTE: This functions is responsible for keeping the volumes needed
 *       list
 *       (ebvllist) element of the restore context up to date.
 *
 * UnmarkObject Parameters:
 *
 *   context      (I) - Pointer to the restore context
 *   thisObject   (I) - The restoral object;
 *                      can be a leaf object (
 *                      e.g., a
 *                      file), or a container object (e.g. a
 *                      directory).
 *
 *   BadFilesOnly (I) - allows unmarking ONLY of files of state BADDATA.
 *
 *   descend      (I) - Should unmark operation descend to operate on the
 *                      content of container objects.
 *
 *   BadFilesCount (O) - returns the file count with BADDATA.
 *
 *   fileMarked    (O) - return the total files marked after this unmark
 *
 *   lenMarkedFiles (O) - return the length of files marked after this unmark
 *
 *   dirMarked     (O) - return the total directories marked after this unmark
 *
 *   otherMarked   (O) - return the total "other" files marked after this unmark
 *
 *   progressCB    (I) - pointer to callback function to report progress and
 *                       test for cancellation
 *
 * Note:
 *        The Progress Callback is currently not used. If the need
 *        for this developes, the routines which use this argument
 *        (mark, unmark, submit) must be enhanced.
 *
 ******************************************************************/

eerrno_ty RSTPI_UnmarkObject(restore_context *context,
                             struct RSTRPC_user_restorable_object
                                    *thisObject,
                             const boolean_ty BadFiledOnly,
                             const boolean_ty descend,
                             unsigned long *BadFilesCount,
                             unsigned long *filesMarked,
                             u_hyper *lenMarkedFiles,
                             unsigned long *dirMarked,
                             unsigned long *otherMarked,
                             RSTPI_MarkProgressProc progressCB)
{
    *BadFilesCount=0;
    *filesMarked=0;
    *lenMarkedFiles=ul_to_uh(0);
```

```
    char ** c1=(char **) (thisObject->appData.data);
    *otherMarked=0;

    RestoreNode *rnodep=(RestoreNode *)(*c1);

    if(NULL==rnodep)
    {
        return EP_RB_RECOVER_RN_APP_DATA_NULL;
    }

    RestoreNode *parent=rnodep->getParent();
    if(NULL !=parent && parent->nodeType() == RNC_MTFB)
    {
        rnodep=parent;
    }
    /*
     * MTFB (multi trail file backup) nodes are to be treated as atomic.
     * If the item we are trying to unmark is a child of a MTFB node, then
     * we perform the unmark operation on the entire MTFB node.
     */

    rnodep->unmarkNode(descend,FALSE);

    // We start at the backup node pointer since we want the total
    // for the entire backup, not just this node down (except the
    // badFilesCount, in which case the affected nodes will only
    // be from this level down)

    *filesMarked=rnodep->getBackupNodePtr()->countMarkedNodes(
                                                RNC_ANY_DATATYPE);

    *lenMarkedFiles=rnodep->getBackupNodePtr()->totalMarkedSize();
    // The bad file count is a cound of the bad files encountered
    // (not necessarily unmarked) from the current node down.
    *BadFilesCount=rnodep->countMarkedBadfileNodes(RNC_ANY_DATAFILE);

    return E_SUCCESS;

}
```

```
1046    //
```

```
1047    /**********************************************************
1048    *
1049    * Is Object Markable
1050    *
1051    * This function determines if a restorable object has been
1052    * marked for restoration.
1053                   It is intended to allow the user to determine the
1054    * current restore markings for the restorable objects at the same
1055                                                              object tree
1056    * hierarchy level,
1057    *            i.e. objects that have the same parent restorableObject.
1058    *
1059    * Parameters:
1060    *
1061    * context      (I) - Pointer to the restore context
1058    * thisObject   (
1057    I)     - The restoral object to be checked: can be a leaf object (
1059    *                         (e.g. a file), or a container object (
1058    *                                                  e.g., a directory).
1060    * markable     (O) - boolean to receive the marked(1) / unmarked(
1061                                                         0) result
1063    ***********************************************************/
1064
1065    eerrno_ty RSTPI_IsObjectMarkable(restore_context *context,
1064                        struct RSTRPC_user_restorable_object
1063                                                  *thisObject,
1061                                        boolean_ty *markable)
1066 1  {
1068 1      RestoreNode *rnodep = *(RestoreNode **)(thisObject->appData.data);
1070 1      if (NULL==rnodep)
1071 2      {
1072 2          return EP_RB_RECOVER_RN_APP_DATA_NULL;
1073 1      }
1075 1      *markable=rnodep->isNodeMarkable();
1077 1      return E_SUCCESS;
1079 1  }
```

```
1082    //
```

```
1083    /********************************************************************
1084     * Is Object Marked
1085     *
1086     * This function determines if a restorable object has been
1087     * marked for restoration.
                It is intended to allow the user to determine the
1088     * current restore markings for the restorable objects at the same
                                                                object tree
1089     * hierarchy level,
                i.e. objects that have the same parent restorableObject.
1090     *
1091     *
1092     * Parameters:
1093     * context      (I) - Pointer to the restore context
1094     * thisObject   (
                I) - The restoral object to be checked; can be a leaf object (
                            (e.g. a file), or a container object (
1095     *                                                       e.g., a directory).
1096     * marked       (O) - boolean to receive the marked(1) / unmarked(
                                                                    0) result
1097     ********************************************************************/

1099    eerrno_ty RSTPI_IsObjectMarked(restore_context *context,
1100                                   struct RSTRPC_user_restorable_object
1101                                                       *thisObject,
                                        boolean_ty *marked)
1102  1 {

1104  1   RestoreNode *rnodep = *(RestoreNode **)(thisObject->appData.data);

1106  1   if(NULL==rnodep)
1107  2     {
1108  2     return EP_RB_RECOVER_RN_APP_DATA_NULL;
1109  1     }

1111  1   *marked=rnodep->isNodeMarked();

1113  1   return E_SUCCESS;

1115    }
```

```
1117
1118      //
```

```
1118  1   /*****************************************************************
1119  1    *
1120  1    * Get All Backup Times
1121  1    *
1122  1    * Function Description:
1123  1    *    Retrieve the dates of the backups within the time range
1124  1    *    specified by the caller.
1125  1    *
1126  1    * Parameters:
1127  1    *    context      - (I) Pointer to the restore context
1128  1    *    startTime    - (I) include no earlier than this date
1129  1    *    endTime      - (I) include no later than this date
1130  1    *    flags        - (I) flags - complete/partial
1131  1    *    timesList    - (O) ptr to linked list of times
1132  1    *    numEntries   - (O) count of times returned
1133  1    *
1134  1    * Return Codes:
1135  1    *    E_SUCCESS    - operation completed successfully
1136  1    *
1137  1    ******************************************************************/
      
1139      eermo_ty RSTPI_GetAllBackupTimes(restore_context *context,
1140                                       const time_t startTime,
1141                                       const time_t endTime,
1142                                       RSTRPC_backup_flags_ty flags,
1143                                       RSTRPC_time_list **timesList,
1144                                       short *numEntries)
1145  1   {
      
1147  1       *timesList=NULL;
      
1149  1       //
1150  1       // Range
1151  1       // Now look through current backups and put on list if in
1152  1       //
1153  1       struct restoreContextData *rcd=(struct restoreContextData *)(
                   context->appData);
      
1155  1       if(!rcd)
1156  2       {
1157  2           return EP_RB_RECOVER_RC_APP_DATA_NULL;
1158  1       }
      
1160  1       WiSetNode *wilist=rcd->currentWisetNode;
      
1162  1       if(!wilist)
1163  2       {
1164  2           return EP_RB_RECOVER_RN_APP_DATA_NULL;
1165  1       }
      
1168  1       int first=1;
      
1170  1       RSTRPC_time_list *newlist=NULL;
      
1172  1       *numEntries=0;       // Count of number of entries found
      
1174  1       for(RestoreNode *bn=wilist->getFirstChild();bn;
                   bn=wilist->getNextChild())
1175  2       {
1176  2           RnPropertyDate *pr=(RnPropertyDate *)(bn->getProperty(
                       PROPERTY_BACKUP_DATE));
1177  2           if(!pr)
1178  3           {
1179  3               continue;
```

```
1180  2      }
1181  2      // Get Unix flavored time
1183  2      DateTime *dt=(DateTime *)(pr->getValue());
1185  2      time_t backupTime=dt->unixTime();
1187  2      // time_t backupTime=(DateTime)(pr->getValue())->unixTime();
1188  2      //
1189  2      // Get all dates if start and end time are zero
1190  2      // otherwise make sure the time is in the selected range
1191  2      //
1192  2      // Also, get complete backups ONLY if that is what is required
1193  2      //
1194  2      //
1195  2      if (
1196  2          ( ( (startTime==0) && (endTime==0) ) ||
1197  2            (startTime<=backupTime) && (endTime>=backupTime)
1198  2          )
1199  2          &&
1200  2          ( (flags==BACKUP_SELECTION_FLAG_PARTIAL_OK) ||
1201  2            bn->getStateBit(STATE_COMPLETE)
1202  2          )
1203  2
1204  3      {
1205  3          //
1206  3          // Ok, we have found a backup time within the
1207  3          // range.  Add it here.
1208  3          //
1209  3          RSTRPC_time_list *newtime=(RSTRPC_time_list *)malloc(
1210  3                  sizeof(struct RSTRPC_time_list));
1211  4          if(!newtime)
1212  4          {
1213  4              rbe_log_stats(0,"plugin.cc - malloc failure");
1214  3              return EP_RB_RECOVER_MALLOC_FAILURE;
1216  3          }
1217  3          //
1218  3          // Increment count of valid time entries
1219  3          //
1221  3          (*numEntries)++;
1222  3          newtime->next=NULL;
1223  3          newtime->time=backupTime;
1224  4          if(newlist)
1225  4          {
1226  3              newlist->next=newtime;
1227  3          }
1229  3          newlist=newtime;
1230  4          if(first)
1231  4          {
1232  4              *timesList=newlist;
1233  3              first=0;
1233  3          }
1235  2      }
1236  1  }
1238  1  return E_SUCCESS;
1239  }
```

```
1241  //
```

```
1242   /****************************************************************
1243    *
1244    * RSTSL_GetCurrentBackupTime
1245    *
1246    *
1247    * Function Description:
             Retrieve the time of the backup that the current restore
                                                            context
1248    * is set to and return it in the preallocated buffer.
1249    *
1250    * Parameters:
1251    *    context    - (I) Pointer to the restore context
1252    *    bkupTime - (O) the time of the backup
1253    *
1254    * Return Codes:
1255    *    E_SUCCESS        - operation completed successfully
1256    *    EP_RB_RECOVER_INVALOP    - call issued out of sequence
1257    *    EP_RB_RECOVER_BAD_ARGS   - invalid input argument
1258    *    EP_RB_RECOVER_NO_CURR_BACKUP   - no valid backup currently
1259    *
1260    *
1262    ****************************************************************/
1263
1264   eerrno_ty RSTPI_GetCurrentBackupTime(restore_context *context,
                                            time_t *backupTime)
1265   {
1266 2    struct restoreContextData *rcd=(struct restoreContextData *)(
                                            context->appData);
1267 1    //
1268 1    // Range
1269 1    //
           // Now look through current backups and put on list if in
1271 1    if(!rcd)
1272 2    {
1273 2        return EP_RB_RECOVER_RC_APP_DATA_NULL;
1274 1    }
1276 1    BackupNode *currentBackup=rcd->currentBackupNode;
1277 1    if(!currentBackup)
1278 2    {
1279 2        return EP_RB_RECOVER_RN_APP_DATA_NULL;
1280 1    }
1282 1    RnPropertyDate *pr=(RnPropertyDate *)(currentBackup->getProperty(
                                     PROPERTY_BACKUP_DATE));
1283 1    if(!pr)
1284 2    {
1285 2        *backupTime=0;
1286 2        return EP_RB_RECOVER_NO_BACKUPTIME;
1287 1    }
1289 1    DateTime *dt=(DateTime *)(pr->getValue());
1290 1    *backupTime=dt->unixTime();
1292 1    return E_SUCCESS;
1293   }
```

```
1297   //
         //
```

```
1298  /****************************************************************
1299   *
1300   * Set Backup For Time
1301   *
1302   * Function Description:
1303   *   Switch to the backup plane of the specified time,
1304   *   that is before the specified time, or the most recent
1305   *   if an exact mactch is not possible.
1306   *
1307   * Parameters:
1308   *   context (I) - Pointer to the restore context
1309   *   forTime (I) - The time for which the backup is requested
1310   *   flags   (I) - Backup constraint flags: e.g., full-only/partial-ok
1311   *
1312   * Return Codes:
1313   *   E_SUCCESS         - operation completed successfully
1314   *   EP_RB_RECOVER_xxx - backup plane cannot be found
1315   *
1316   ****************************************************************/
1317
1318
1319  eerrno_ty RSTPI_SetBackupForTime(restore_context *context,
1320                                    const time_t forTime,
1321                                    RSTRPC_backup_flags_ty flags)
1322  {
1323      struct restoreContextData *rcd=(struct restoreContextData *)(
1324                                      context->appData);
1325      if(!rcd)
1326      {
1327          return EP_RB_RECOVER_RC_APP_DATA_NULL;
1328      }
1329
1330      WiSetNode *wilist=rcd->currentWisetNode;
1331
1332      if(!wilist)
1333      {
1334          return EP_RB_RECOVER_RN_APP_DATA_NULL;
1335      }
1336
1337
1338      //
1339      // Loop through each one on the current list and return the one which we
1340      // are looking for.
1341      //
1342      for(RestoreNode *bn=wilist->getFirstChild();bn;
1343              bn=wilist->getNextChild())
1344      {
1345          RnPropertyDate *pr=(RnPropertyDate *)(bn->getProperty(
1346                              PROPERTY_BACKUP_DATE));
1347          if(!pr)
1348          {
1349              continue;
1350          }
1351          // Get Unix flavored time
1352          DateTime *dt=(DateTime *)(pr->getValue());
1353          time_t backupTime=dt->unixTime();
1354
1355          if ( (forTime >= backupTime) &&
1356               (
```

```
1357               flags==BACKUP_SELECTION_FLAG_PARTIAL_OK) || bn->getStateBit(
1358                                                           STATE_COMPLETE) )
1359          {
1360
1361              BackupNode *bnode=(BackupNode *)bn;
1362              //
1363              // If we already have a current backup node which differs from the
1364              // current one, clear marks.
1365              if(
1368              NULL != rcd->currentBackupNode && rcd->currentBackupNode != bnode)
1369              {
1370                  rcd->currentBackupNode->unmarkNode(TRUE,FALSE);
1371                  rcd->currentBackupNode->unlockWorkitems(context);
1372              }
1373              if(NULL!=rcd->currentBackupNode)
1374                  rcd->currentBackupNode=bnode;
1375              if(rcd->currentBackupNode->lockWorkitems(context))
1376              {
1377                  return EP_RB_RECOVER_WI_LOCKED;
1378              }
1379              // We found the one to to backup
1380              bnode->populate(context,POPULATE_CHILDREN);
1381              rcd->currentBackupNode=bnode;
1382              return E_SUCCESS;
1383          }
1384      }
1385      // Commented out to leave current backup unchanged if no call found
1387      // rcd->currentBackupNode=NULL;
1389      return E_RB_RECOVER_NO_BACKUP_FOUND;
1391  }
```

```
1394

//
```

```
1395  /*************************************************************************
1396   *
1397   * Set Previous Backup
1398   *
1399   * Function Description:
1400   *    Set the restore context to that of the previous backup with
1401   *    respect
1402   *    to the current one.
1403   *
1404   * Parameters:
1405   *    context  (I)  - Pointer to the restore context
1406   *    flags    (I)  - Backup constraint flags: e.g., full-only/partial-ok
1407   *
1408   * Return Codes:
1409   *    E_SUCCESS              - operation completed successfully
1410   *    EP_RB_NO_PREV_CATALOG  - when at the first catalog
1411   *    EP_RB_RECOVER_PERMISSION_DENIED - when user cannot access the
1412   *                           file
1413   *                           of the new catalog
1414   *
1415   *************************************************************************/
1416
1417  eerrno_ty RSTPI_SetPrevBackup(restore_context *context,
1418                                RSTRPC_backup_flags_ty flags)
1419  {
1420      struct restoreContextData *rcd=(struct restoreContextData *)(
1421                                      context->appData);
1422
1423      WiSetNode *wilist=rcd->currentWisetNode;
1424
1425      if(!wilist)
1426      {
1427          return EP_RB_RECOVER_RC_APP_DATA_NULL;
1428      }
1429
1430      BackupNode *bnode=rcd->currentBackupNode;
1431
1432      if(!bnode)
1433      {
1434          return EP_RB_RECOVER_RC_APP_DATA_NULL;
1435      }
1436
1437      //
1438      // Set the iterator to the current item
1439      //
1440
1441      if(!wilist->setIterToSpecificChild(rcd->currentBackupNode))
1442      {
1443          return EP_RB_RECOVER_SETTING_TO_SPECIFIC_CHILD;
1444      }
1445
1446      bool first=1;
1447      for(RestoreNode *rn=wilist->getThisChild();rn;
1448                       rn=wilist->getPrevChild())
1449      {
1450          if(first)
1451          {
1452              // Skip this node so we don't get self
1453              first=0;
1454          }
1455          else
1456          {
```

```
1457  3    if ( (
1458  4    flags==BACKUP_SELECTION_FLAG_PARTIAL_OK)  || rn->getStateBit(
                                                         STATE_COMPLETE))
1460  4    {
           // If we already have a current backup node,
           //                                            clear all marks
1461  4        if(rcd->currentBackupNode)
1462  5        {
1463  5            rcd->currentBackupNode->unmarkNode(TRUE,FALSE);
1464  4        }
1465  4        if(NULL!=rcd->currentBackupNode)
1466  5        {
1467  5            rcd->currentBackupNode->unlockWorkitems(context);
1468  4        }
1469  4        BackupNode *bnode=(BackupNode *)rn;
1470  4        rcd->currentBackupNode=bnode;
1471  4        if(!rcd->currentBackupNode->lockWorkitems(context))
1472  5        {
1473  5            return EP_RB_RECOVER_WI_LOCKED;
1474  4        }
1475  4        bnode->populate(context,POPULATE_CHILDREN);
1476  4        // We found the one to backup
1477  4        return E_SUCCESS;
1478  3    }
1479  2    }
1480  1    }
1482  1    return EP_RB_RECOVER_NONE_EXISTS;
1483  1    }
```

```
1487       //
```

```
1488      /***************************************************************************
1489      *
1490      * Set Next Backup
1491      *
1492      * Function Description:
1493   1  * This routine must set the restore environment to the the next
1494   1  * of the current top level object.
1495      *
1496      * Parameters:
1497   2  *     context  (I) - Pointer to the restore context
1498   2  *     flags    (
1499   3  *              I) - Backup constraint flags: e.g., full-only/partial-ok
1500      * Return Codes:
1501   2  *     E_SUCCESS               - operation completed successfully
1502   2  *     EP_RB_RECOVER_NO_NEXT_CATALOG - when at the most recent
1503   2  *                                     catalog
1504   2  *     EP_RB_RECOVER_PERMISSION_DENIED - when user cannot access the
1505      *                                       file
1506      *     EP_RB_RECOVER_NO_CATALOG  - when mcat_set_mcplane failed
1507      *                                 of the new catalog
1509      ***************************************************************************/
1510      eerrno_ty RSTPI_SetNextBackup(restore_context *context,
1511                                    RSTRPC_backup_flags_ty flags)
1513      {
1516   1      struct restoreContextData *rcd=(struct restoreContextData *)(
1516   1                                     context->appData);
1518   1      WiSetNode *wilist=rcd->currentWisetNode;
1519   2
1520   2      if(!wilist)
1521   1      {
1524   1          return EP_RB_RECOVER_RC_APP_DATA_NULL;
1526   1      }
1527   2
1528   2      BackupNode *bnode=rcd->currentBackupNode;
1529   1
1532   1      if(!bnode)
1535   1      {
1536   1          return EP_RB_RECOVER_RC_APP_DATA_NULL;
1537   2      }
1538   2
1539   3      RestoreNode *lastprev=NULL;
1540   3
1541   3      RestoreNode *rn=wilist->getFirstChild();
1542   2      while(rn && rn != bnode)
1543   1      {
1546   1          if( flags==BACKUP_SELECTION_FLAG_PARTIAL_OK ||
1546   1              rn->getStateBit(STATE_COMPLETE))
                    {
                        lastprev=rn;
                    }
                    rn=wilist->getNextChild();
                }

                if(lastprev)
```

```
1547   2      {
1548   2          // If we already have a current backup node, clear all marks
1549   2          if(rcd->currentBackupNode)
1550   3          {
1552   3              rcd->currentBackupNode->unmarkNode(TRUE,FALSE);
1553   2          }
1554   2          if(NULL!=rcd->currentBackupNode)
1555   2          {
1556   3              rcd->currentBackupNode->unlockWorkitems(context);
1557   2          }
1558   2          BackupNode *bnode=(BackupNode *)lastprev;
1559   2          rcd->currentBackupNode=bnode;
1560   2          if(!rcd->currentBackupNode->lockWorkitems(context))
1561   3          {
1562   3              return EP_RB_RECOVER_WI_LOCKED;
1563   2          }
1564   2          // We found the one to backup
1565   2          bnode->populate(context, POPULATE_CHILDREN);
1567   2      }
1568   1      return E_SUCCESS;
1570   1  }
1571      return EP_RB_RECOVER_NONE_EXISTS;
```

1578

//

```
1579  /********************************************************************
1580   *
1581   * Set Most Recent Backup
1582   *
1583   * Function Description:
1584   *   Set the restore context to that of the most recent backup
1585   *   plane.
1586   *
1587   * Parameters:
1588   *   context  (I) - Pointer to the restore context
1589   *   flags    (I) - Backup constraint flags: e.g., full-only/partial-ok
1590   *a
1591   * Return Codes:
1592   *   E_SUCCESS        - operation completed successfully
1593   *   EP_RB_RECOVER_PERMISSION_DENIED - when user cannot access the
1594   *                                      file of
1595   *                                      the new catalog.
1596   *
1597   *
1599   ******************************************************************/
1600
1601  eerrno_ty RSTPI_SetMostRecentBackup(restore_context *context,
                                          RSTPC_backup_flags_ty flags)
1602  {
1603     struct restoreContextData *rcd=(struct restoreContextData *)(
                                          context->appData);
1606     WiSetNode *wilist=rcd->currentWisetNode;
1608     if(!wilist)
1609     {
1610        return EP_RB_RECOVER_RC_APP_DATA_NULL;
1611     }
1613     RestoreNode *bnode=NULL;
1615     //
1616     // Set the iterator to the curent item
1617     //
1619     for(RestoreNode *rn=wilist->getFirstChild();rn;
                         rn=wilist->getNextChild())
1620     {
1621        if( flags==BACKUP_SELECTION_FLAG_PARTIAL_OK || rn->getStateBit(
                       STATE_COMPLETE))
1622        {
1623           // If we already have a current backup node,
1624           if(rcd->currentBackupNode)
1625           {
1626              rcd->currentBackupNode->unmarkNode(TRUE,FALSE);
1627           }
1628           if(NULL != rcd->currentBackupNode)
1629           {
1630              if(rn==rcd->currentBackupNode)
1631              {
1632                 /* We are already at the right place so return
                       success */
1633                 return E_SUCCESS;
1634              }
1635              rcd->currentBackupNode->unlockWorkitems(context);
```

```
1636  3          }
1637  3          BackupNode *bnode=(BackupNode *)rn;
1638  3          rcd->currentBackupNode=bnode;
1639  3          if(!rcd->currentBackupNode->lockWorkitems(context))
1640  4          {
1641  4              return EP_RB_RECOVER_WI_LOCKED;
1642  3          }
1643  3          bnode->populate(context, POPULATE_CHILDREN);
1644  3          // We found the one to backup
1646  3          return E_SUCCESS;
1647  2      }
1648  1  }
1651  1  return EP_RB_RECOVER_SET_BUTIME_ERROR;
1652     }
```

```
1657  //
```

```
1659  /****************************************************************
1660   *
1661   * Set First Backup
1662   *
1663   * Function Description:
1664   *    Set the restore context to that of the first backup plane.
1665   *
1666   *
1667   * Parameters:
1668   *    context (I) - Pointer to the restore context
1669   *    flags   (
1670   *    I) - Backup constraint flags: e.g., full-only/partial-ok
1671   *
1672   * Return Codes:
1673   *    E_SUCCESS      - operation completed successfully
1674   *    EP_RB_RECOVER_PERMISSION_DENIED - when user cannot access the
1675   *                          the new catalog
1676   *                          file of
1678   *
1679   *
1680   ****************************************************************/
1682  struct restoreContextData *rcd=(struct restoreContextData *)(
                                      context->appData);
1685  WiSetNode *wilist=rcd->currentWisetNode;
1687  if(!wilist)
1688  {
1689      return EP_RB_RECOVER_RC_APP_DATA_NULL;
1690  }
1692  //
1693  // Set the iterator to the current item
1694  //
1696  for(RestoreNode *rn=wilist->getFirstChild();rn;
                       rn=wilist->getNextChild())
1697  {
1698      if( flags==BACKUP_SELECTION_FLAG_PARTIAL_OK || rn->getStateBit(
                                       STATE_COMPLETE))
1699      {
1700          // If we already have a current backup node,
                                               clear all marks
1701          if(rcd->currentBackupNode)
1702          {
1703              rcd->currentBackupNode->unmarkNode(TRUE,FALSE);
1704          }
1705          if(NULL!=rcd->currentBackupNode)
1706          {
1707              rcd->currentBackupNode->unlockWorkitems(context);
1708          }
1709          BackupNode *bnode=(BackupNode *)rn;
1710          rcd->currentBackupNode=bnode;
1711          if(!rcd->currentBackupNode->lockWorkitems(context))
1712          {
1713              return EP_RB_RECOVER_WI_LOCKED;
1714          }
1715          bnode->populate(context, POPULATE_CHILDREN);
```

```
1717          return E_SUCCESS;
1718      }
1719  }
1721  return EP_RB_RECOVER_SET_BUTIME_ERROR;
1722  }
```

```
1728

         //
```

```
1730    /****************************************************************
1731    *
1732    * Is There Prev Backup
1733    *
1734    * Function Description:
1735    *     Determine if a backup exists prior to the backup that is
1736    *     currently selected.
1737    *
1738    * Parameters:
1739    *     context  (I) - Pointer to the restore context
1740    *     flags    (I) - Backup constraint flags: e.g., full-only/partial-ok
1741    *     isThere  (O) - TRUE/FALSE that requested backup does exist
1742    *
1743    * Return Codes:
1744    *     E_SUCCESS          - operation completed successfully
1745    *     EP_RB_RECOVER_xxx  - when errors occur accessing
                                    catalogs
1746    *
        ****************************************************************
        */
1748    eerrno_ty RSTPI_IsTherePrevBackup(restore_context *context,
1749                                      RSTRPC_backup_flags_ty flags,
1750                                      boolean_ty *isThere)
1751    {
1753        struct restoreContextData *rcd=(struct restoreContextData *)(
                                            context->appData);
1755        *isThere=FALSE;
1757        WiSetNode *wilist=rcd->currentWisetNode;
1759        if(!wilist)
1760        {
1761            return EP_RB_RECOVER_RC_APP_DATA_NULL;
1762        }
1765        BackupNode *bnode=rcd->currentBackupNode;
1767        //Debuggine only
1768        RestoreNode *rob=bnode;
1770        if(!bnode)
1771        {
1772            return EP_RB_RECOVER_RC_APP_DATA_NULL;
1773        }
1775        //
1776        // Set the iterator to the curent item
1777        //
1779        if(!wilist->setIterToSpecificChild(bnode))
1780        {
1781            return EP_RB_RECOVER_SETTING_TO_SPECIFIC_CHILD;
1782        }
1784        bool first=TRUE;
1786        for(RestoreNode *rn=wilist->getThisChild());rn;
                                                       rn=wilist->getPrevChild())
1787        {
1788            if(first)
```

```
1789  3              {
1790  3              first=FALSE;
1791  2              }
1792  2          else
1793  3              {
1794  3              if(
                     flags==BACKUP_SELECTION_FLAG_PARTIAL_OK || rn->getStateBit(
                                                                STATE_COMPLETE))
1795  4                  {
1796  4                  *isThere=TRUE;
1797  4                  return E_SUCCESS;
1798  3                  }
1799  2              }
1800  1          }
1802  1      return E_SUCCESS;
1803  1      }
```

```
1807  //
```

```
1808  /*****************************************************************
1809   *
1810   * Is There Next Backup
1811   *
1812   * Function Description:
1813   *    Determine if a backup exists after the backup that is
1814   *    currently selected.
1815   *
1816   * Parameters:
1817   *    context   (I) - Pointer to the restore context
1818   *    flags     (I) - Backup constraint flags: e.g., full-only/partial-ok
1819   *    isThere   (O) - TRUE/FALSE that requested backup does exist
1820   *
1821   * Return Codes:
1822   *    E_SUCCESS             - operation completed successfully
1823   *    EP_RB_RECOVER_XXX     - when errors occur accessing
                                   catalogs
1824   *
1826   *****************************************************************/
1827
1828  eerrno_ty RSTPI_IsThereNextBackup(restore_context *context,
                                       RSTRPC_backup_flags_ty flags,
1829                                     boolean_ty *isThere)
1830  {
      struct restoreContextData *rcd=(struct restoreContextData *)(
                                       context->appData);

1833      *isThere=FALSE;

1835      WiSetNode *wilist=rcd->currentWisetNode;

1837      if(!wilist)
1838      {
1839          return EP_RB_RECOVER_RC_APP_DATA_NULL;
1840      }

1843      BackupNode *bnode=rcd->currentBackupNode;

1845      if(!bnode)
1846      {
1847          return EP_RB_RECOVER_RC_APP_DATA_NULL;
1848      }

1852      RestoreNode *rn=wilist->getFirstChild();
1853      while(rn && rn != bnode)
1854      {
1855          if( flags==BACKUP_SELECTION_FLAG_PARTIAL_OK ||
                  rn->getStateBit(
                  STATE_COMPLETE))
1856          {
1857              *isThere=TRUE;
1858              return E_SUCCESS;
1859          }
1860          rn=wilist->getNextChild();
1861      }

1864      return E_SUCCESS;
1866  }
```

```
1869  /*****************************************************************
1870   *
1871   * Is Object Markable
1872   *
1873   * Function Description:
1874   *    Returns TRUE if the specified object is markable by the user,
1875   *    returns FALSE if it is not. This function applies only to
1876   *    container (directory) and leaf (file) objects.
1877   *
1878   * Parameters:
1879   *    context          - (I) Pointer to the restore context
1880   *    thisObject       - (I) ptr to the restorableObject in question
1881   *
1882   * Return:
1883   *    TRUE  - the specified object is markable by the user;
1884   *    FALSE - the specified object is not markable by the user;
1885   *
1886   *****************************************************************/
1887
1888  boolean_ty
1889  RSTPI_IsObjectMarkable(
1890                restore_context                        *context,
1891                struct RSTRPC_user_restorable_object   *thisObject )
1892  {

1894      char **cl=(char **)(thisObject->appData.data);
1895      RestoreNode *rnodep=(RestoreNode *)(*cl);
          return rnodep->isNodeMarkable();

          return E_SUCCESS;
      }
```

```
1898  /*****************************************************************
1899   *
1900   * Get Top Level Templates:
1901   *
1901   * This function is required to retrieve the templates with which a
             backup
1902   * object could have been backed up.
1903   *
1904   * Parameters:
1905   *    context        (I)   - Pointer to the restore context
1906   *    topLevelObj    (I)   - the top level object
1907   *    templates      (O)   - pointer to receive the start of the list of templates
1908   *    numberEntries  (O)   - the real number of templates returned in the list
1909   *
1910   *
              *****************************************************************/
1912  eerrno_ty  RSTPI_GetTopLevelTemplates(
                      restore_context              *context,
                      struct RSTRPC_top_level_obj  *topLevelObj,
                      struct RSTRPC_name_list      **templates,
                      short                        *numberEntries )
1913  {
1914      char ** c1=(char **)(topLevelObj->appData.data);
1915      RestoreNode *rnodep=(RestoreNode *)(*c1);
1916
1917      if(NULL==rnodep)
1918      {
1920          rbe_log_stats(
1921              0,"RSTPI_GetTopLevelTemplates - Mark object has no app data");
1922          return EP_RB_RECOVER_RN_APP_DATA_NULL;
1923      }
1924
1926      //
1927      // Since we are setting a top level object,
1928      // the marks on any existing
1929      // object must be cleared.
1930      //
1931      struct restoreContextData *rcd=(struct restoreContextData *)(
                                         context->appData);
1932
1933      if(NULL!=rcd)
1934      {
1935          if(NULL!=rcd->currentBackupNode)
1936              rcd->currentBackupNode->unmarkNode(TRUE,FALSE);
1937      }
1940      *numberEntries=NULL;
1941      *templates=NULL;
1944      for(RestoreNode *rn=rnodep->getFirstChild();
1945          rn;
1946          rn=rnodep->getNextChild())
1947      {
1949          if(rn->nodeType() != RNC_BACKUP)
1950              continue;
```

```
1954      RnProperty *tp=rn->getProperty(PROPERTY_TEMPLATE);
1955      if(NULL==tp)
1956      {
1957          rbe_log_stats(
1958              0,"RSTPI_GetTopLevelTemplates: Template property not found on backup
                      node");
1959          continue;
1961      }
1962
1964      RnPropertyChar *cp=(RnPropertyChar *)tp;
1965      char * tname=(char *)(cp->getValue());
1966
1967      //
1969      // We should always find a template property.  Skip this item
1970      // if we don't.
1971      //
1972      if(tname[0] == '\0')
1973      {
1975          rbe_log_stats(
1977              0,"RSTPI_GetTopLevelTemplates: Template property ob backup node has no
                      value");
1979          continue;
1980      }
1982      // Now, let's see if we already have it...
1983      struct RSTRPC_name_list *lookat_next;
1984      struct RSTRPC_name_list *lookat_entry=*templates;
1985      bool addIt=TRUE;
1986      while(lookat_entry)
1987      {
1988          lookat_next=lookat_entry->next;
1989          if(0==strcmp(tname,lookat_entry->name))
1990          {
1991              lookat_next=NULL;
1993              addIt=FALSE;    // Fount it!!!
1994          }
1995          lookat_entry=lookat_next;
1996      }
1997
1998      //
1999      // Add the entry only if it hasn't already been placed on the
2000      //   list
2001      //
2003      if(addIt)
2005      {
2006          //
2007          // Add entry to the list if not already here...
2008          //
2009          struct RSTRPC_name_list *entry;
2011          entry=(struct RSTRPC_name_list *)malloc(sizeof(
                                          struct RSTRPC_name_list));
2012          if(NULL==entry)
              {
                  rbe_log_stats(
                      0,"RSTPI_GetTopLevelTemplates: Malloc failure");
                  return EP_RB_RECOVER_MALLOC_FAILURE;
              }
              entry->next=*templates;
              entry->name=es1_strdup(tname);
```

```
2013  3          if(NULL == entry->name)
2014  4          {
2015  4              rbe_log_stats(
2016  3                  0, "RSTPI_GetTopLevelTemplates: Malloc failure");
2017  3              return EP_RB_RECOVER_MALLOC_FAILURE;
2019  3          }
2021  3          *templates=entry;
2022  2          (*numberEntries)++;
2024  1      }
        }
2027  1      return E_SUCCESS;
2028    }
```

```
2031    /*****************************************************************
2032     *
2033     * Clear Restore Context :
2034     *
2035     * This function is called to notify the plug-in that its currently
2036     *                                                         selected
2037     * top level object is no longer selected (
2038     *                                    for browsing and marking). The
2039     * plug-in should perform whatever cleanup and memory deallocation is
2040     * appropriate.
2041     *
2042     * Returns:
2043     *     E_SUCCESS          on success
2044     *     EP_RB_RECOVER_xxx  on error
2045     *
2046     * Parameters:
2047     *     context      (I)  -  Pointer to the restore context
2048     *
2049     *****************************************************************/
2050    eerrno_ty RSTPI_ClearRestoreContext( restore_context *context )
2051  1 {
2052  1     struct restoreContextData *rcd=(struct restoreContextData *)(
                                                        context->appData)
2053  1     ;
2054  1     rbe_log_debug(0, "In RSTPI_ClearRestoreContext");
2056  1     if(NULL == rcd)
2057  2     {
2058  2         return EP_RB_RECOVER_RC_APP_DATA_NULL;
2059  1     }
            //
            // Get the restore context so we can find the app data
            //
2061  1     if(NULL != rcd->currentBackupNode)
2062  1     {
2063  2         rbe_log_debug(0, "RSTPI_ClearRestoreContext: unlocking %s",
2064  2                     rcd->currentBackupNode->getNameChar());
2065  2         rcd->currentBackupNode->unlockWorkitems(context);
2066  2         rcd->currentBackupNode->unmarkNode(TRUE,FALSE);
2067  2     }
2068  1     rcd->currentBackupNode=NULL;
2069  1     // Clear locks if they exist
2071  1     return E_SUCCESS;
2072    }
```

```
2075  /*******************************************************************
2076   *
2077   * Is There Prev Backup For Time
2078   *
2079   * Function Description:
2080   *    Determine if a backup exists prior to the specified time
2081   *
2082   * Parameters:
2083   *    context (I) - Pointer to the restore context
2084   *    thisTime(I) - Time for the query
2085   *    flags    (I) - Backup constraint flags: e.g., full-only/partial-ok
2086   *    isThere (O) - TRUE/FALSE that requested backup does exist
2087   *
2088   * Return Codes:
2089   *    E_SUCCESS      - operation completed successfully
2090   *    EP_RB_RECOVER_xxx   - when errors occur accessing catalogs
2091   *
      *******************************************************************
      */
2093  eerrno_ty RSTPI_IsTherePrevBackupForTime( restore_context *context,
2094                                             const time_t    thisTime,
2095                                             RSTRPC_backup_flags_ty flags,
2096                                             boolean_ty     *isThere )
2097  {
2098      struct restoreContextData *rcd=(struct restoreContextData *)(
                                           context->appData);
2100      *isThere=FALSE;

2103      WiSetNode *wilist=rcd->currentWisetNode;

2105      if(!wilist)
2106      {
2107          return EP_RB_RECOVER_RC_APP_DATA_NULL;
2108      }

2111      BackupNode *bnode=rcd->currentBackupNode;

2113      if(!bnode)
2114      {
2115          return EP_RB_RECOVER_RC_APP_DATA_NULL;
2116      }

2119      //
2120      // Loop over all backup objects in this set looking for one which is
      //                                                                previous
2121      // to this one.
2122      //
2123      for(RestoreNode *rn=wilist->getFirstChild();rn;
                                            rn=wilist->getNextChild())
2124      {
2125          if((flags==BACKUP_SELECTION_FLAG_PARTIAL_OK) || rn->getStateBit(
                                                            STATE_COMPLETE))
2126          {
2127              BackupNode *bn=(BackupNode *)rn;
2128              RnPropertyDate *dp=(RnPropertyDate *)(rn->getProperty(
                                                PROPERTY_BACKUP_DATE));
2129              time_t btime=dp->getValue()->unixTime();
```

```
2131              if(thisTime > btime)
2132              {
2133                  *isThere=TRUE;
2134                  return E_SUCCESS;
2135              }
2136          }
2137      }

2139      return E_SUCCESS;

2141  }
```

```
2143  /****************************************************************
2144   *
2145   *
2146   *  Is There NextBackup For Time
2147   *
2148   *  Function Description:
2149   *      Determine if a backup exists after to the specified time
2150   *
2151   *  Parameters:
2152   *      context (I)  - Pointer to the restore context
2153   *      thisTime(I)  - Time for the query
2154   *      flags   (I)  - Backup constraint flags: e.g., full-only/partial-ok
2155   *      isThere (O)  - TRUE/FALSE that requested backup does exist
2156   *
2157   *  Return Codes:
2158   *      E_SUCCESS            - operation completed successfully
2159   *      EP_RB_RECOVER_xxx    - when errors occur accessing catalogs
2160   *
2161   */
2162  eerrno_ty RSTPI_IsThereNextBackupForTime( restore_context      *context,
2163                                            const time_t          thisTime,
2164                                            RSTPC_backup_flags_ty flags,
2165                                            boolean_ty           *isThere )
2166  {
2168      struct restoreContextData *rcd=(struct restoreContextData *)(
                                          context->appData);

2171      *isThere=FALSE;

          WiSetNode *wilist=rcd->currentWiSetNode;
2173      if(!wilist)
2174      {
2175          return EP_RB_RECOVER_RC_APP_DATA_NULL;
2176      }

2179      BackupNode *bnode=rcd->currentBackupNode;
2181      if(!bnode)
2182      {
2183          return EP_RB_RECOVER_RC_APP_DATA_NULL;
2184      }

2187      //
2188      // Loop over all backup objects in this set looking for one which is
2189      // to this one.                                            previous
2190      //
2191      for(RestoreNode *rn=wilist->getFirstChild();rn;
                           rn=wilist->getNextChild())
2192      {
2193          if((flags==BACKUP_SELECTION_FLAG_PARTIAL_OK) ||
                             rn->getStateBit(STATE_COMPLETE))
2194          {
2195              BackupNode *bn=(BackupNode *)rn;
2196              RnPropertyDate *dp=(RnPropertyDate *)(rn->getProperty(
                                     PROPERTY_BACKUP_DATE));
2197              time_t btime=dp->getValue()->unixTime();

2199              if(thisTime < btime)
```

```
2200              {
2201                  *isThere=TRUE;
2202                  return E_SUCCESS;
2203              }
2204          }
2205      }

2207      return E_SUCCESS;

2209  }
```

```
2211  /**********************************************************
2212  * Identify:
2213  *
2214  * This function is called once,
2215  *         to identify and validate the plug-in with
2216  *         regard to the operating restore engine.
2217  *         The version number is checked
2218  *         for compatibility with the restore engine,
2219  *         and the optional features
2220  *         of the plug-in are specified.
2221  *
2222  * Parameters:
2223  *   pi_defs (
2224  *      O)  -  address of the structure identifying the plugin to the
2225  *                restore engine. Its fields consist of:
2226  *
2227  *   version  -  Version of the plugin header that the library was
2228  *                                                        built with
2229  *
2230  *   name     -  Name of the backup application (
2231  *                                       64 byte buffer address)
2232  *
2233  *   options  -  Bit mask identifying the optional plug-in features
2234  *                                                        supported
2235  *      The bit definitions for this parameter (
2236  *                               RSTPI_OPTION_MASK...) are
2237  *               defined above.
2238  *
2239  *   wi_types  -  pointer to array of workitem types supported by the
2240  *                                                           plugin
2241  *   num_types -  number of witypes in the witypes array
2242  *
2243  * Returns:
2244  *      E_SUCCESS            on success
2245  *      EP_RB_RECOVER_xxx    on error
2246  *
2247  ***********************************************************/
2248  eerrno_ty RSTPI_Identify( const struct pluginIDdata **pi_defs )
2249  {
2250      static char types[2] = {WI_TYPE_DC_NETWORK,
                                  WI_TYPE_DC_WORKER};
         static struct pluginIDdata myPiID = {
                                    /* plugin Identification */
                                    RSTPI_VERSION,
                                    "Symmetrix Connect Restore
                                    Plug-in",
         RSTPI_OPTION_SYMM_RESTORE_ALLOWED,
                                    types,
                                    2
         };

         *pi_defs = &myPiID;

         return E_SUCCESS;
  }
```

```
2253  /**********************************************************
2254  * Finish:
2255  *
2256  * This function is called to allow the plug-in to clean up its
2257  *         internal
2258  *         storage when a restore session is ending.
2259  *
2260  * Parameters:
2261  *   context (I) - Pointer to the restore context
2262  *
2263  * Returns:
2264  *      E_SUCCESS            on success
2265  *      EP_RB_RECOVER_xxx    on error
2266  *
2267  ***********************************************************/
2268  eerrno_ty RSTPI_Finish( restore_context *context )
2269  {
2270      struct restoreContextData *rcd =
                      (struct restoreContextData *)(
                              context->appData);

2272      delete rcd->currentWisetListNode;
2273      delete rcd->currentWisetNode;
2274      delete rcd->currentBackupNode;
2275      free (rcd);

2277      return E_SUCCESS;
2278  }
```

```
2280   /*************************************************************************
2281   * Does Alternate Exist
2282   *
2283   * This routine determines if an alternate trailset exists for the
2284   * given template.
2285   *
2286   * Parameters:
2287   * context      (I)  - Pointer to the restore context
2288   * templateName (I)  - The name of the template to look for
2289   * exists       (
2290   *             O)  - Return flag for whether or not the alternate exists
2291   *
2293   *************************************************************************/
2294   eerrno_ty
2295   RSTPI_DoesAlternateExist( restore_context        *context,
2296                             const template_name_ty templateName,
2297 1                           boolean_ty             *exists )
2298 1  {
2299 1  /************** TEMPORARY,
2300 1     until we get the real version: *********************/
2301     *exists = FALSE;
         return E_SUCCESS;
         }
```

restore_dcpi/plugin.cc 75

Thu Jan 03 12:52:58 2008

Thu Jan 03 12:52:58 2008

restore_dcpi/plugin.cc 76

Thu Jan 03 12:52:58 2008

Thu Jan 03 12:52:58 2008